

HEATHKIT

CONTINUING EDUCATION

INTRODUCTION TO MICROPROCESSORS

MICROPROCESSORS



Educational Systems

Model EC-6800
HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022
595-2400

Copyright © 1980
Second printing-1980
Heath Company
All Rights Reserved
Printed in the United States of America

CONTENTS

Course Objectives	III
Course Outline	III
Unit One — Number Systems and Codes	1-1
Unit two — Microcomputer Basics	2-1
Unit Three — Computer Arithmetic	3-1
Unit Four — Introduction to Programming	4-1
Unit Five — The 6808 Microprocessor — Part 1	5-1
Unit Six — The 6808 Microprocessor — Part 2	6-1
Unit Seven — Programming Experiments	7-1
Appendix A — Definition of the Executable Instructions	A-1
Appendix B — Data Sheets	B-1

COURSE OBJECTIVES

When you have completed this course, you will understand the basic principles of microprocessor operation and be able to program a representative microprocessor.

COURSE OUTLINE

UNIT 1 NUMBER SYSTEMS AND CODES

- I. Introduction
- II. Unit Objectives
- III. Unit Activity Guide
- IV. Decimal Number System
- V. Binary Number System
 - A. Positional Notation
 - B. Converting Between the Binary and Decimal Number System
- VI. Octal Number System
 - A. Conversion from Decimal to Octal
 - B. Converting Between the Octal and Binary Number System
- VII. Hexadecimal Number System
 - A. Converting from Decimal to Hexadecimal
 - B. Converting Between the Hexadecimal and Binary Number Systems
- VIII. Binary Codes
 - A. Binary Coded Decimal
 - B. Special Binary Codes
 - C. Alpha Numeric Codes
- IX. Experiment
- X. Unit Examination
- XI. Examination Answers

UNIT 2 MICROCOMPUTER BASICS

- I. Introduction
- II. Unit Objectives
- III. Unit Activity Guide
- IV. Terms and Conventions
 - A. Stored Program Concept
 - B. Computer Words
 - C. Word Length
- V. An Elementary Microcomputer
 - A. The Microprocessor Unit (MPU)
 - B. Memory
 - C. Fetch — Execute Sequence
 - D. A Sample Program
- VI. Executing a Program
 - A. The Fetch Phase
 - B. The Execute Phase
 - C. Fetching the Add Instruction
 - D. Executing the Add Instruction
 - E. Fetching and Executing the HLT Instruction
- VII. Addressing Modes
 - A. Inherent or Implied Addressing
 - B. Immediate Addressing
 - C. Direct Addressing
 - D. Sample Program Using Direct Addressing
 - E. Executing the Sample Program
 - F. Combining Addressing Modes
- VIII. Experiment
- IX. Unit Examination
- X. Examination Answers

UNIT 3 COMPUTER ARITHMETIC

- I. Introduction
- II. Unit Objectives
- III. Unit Activity Guide
- IV. Binary Arithmetic
 - A. Binary Addition
 - B. Binary Subtraction
 - C. Binary Multiplication
 - D. Binary Division
 - E. Representing Negative Numbers
- V. Two's Complement Arithmetic
 - A. Ten's Complement Arithmetic
 - B. Two's Complement Subtraction
 - C. Arithmetic With Signed Numbers
- VI. Boolean Operations
 - A. AND Operation
 - B. OR Operation
 - C. Exclusive OR Operation
 - D. Invert Operation
- VII. Experiment
- VIII. Unit Examination
- IX. Examination Answers

UNIT 4 INTRODUCTION TO PROGRAMMING

- I. Introduction
- II. Unit Objectives
- III. Unit Activity Guide
- IV. Branching
 - A. Relative Addressing
 - B. Executing a Branch Instruction
 - C. Branching Forward
 - D. Branching Backward
- V. Conditional Branching
 - A. Condition Codes
 - B. Conditional Branch Instructions
- VI. Algorithms
 - A. Multiplying by Repeated Addition
 - B. Dividing by Repeated Subtraction
 - C. Converting BCD to Binary
 - D. Converting Binary to BCD
- VII. Additional Instructions
 - A. Add With Carry (ADC) Instruction
 - B. Subtract With Carry (SBC) Instruction
 - C. Arithmetic Shift Accumulator Left (ASLA) Instruction
 - D. Decimal Adjust Accumulator (DAA) Instruction
- VIII. Experiment
- IX. Unit Examination
- X. Examination Answers

UNIT 5 THE 6808 MICROPROCESSOR — PART 1

- I. Introduction
- II. Unit Objectives
- III. Unit Activity Guide
- IV. Architecture of the 6808 MPU
 - A. Programming Model of the 6808 MPU
 - B. Block Diagram of the 6808 MPU
- V. Instruction Set of the 6808 MPU
 - A. Arithmetic Instructions
 - B. Data Handling Instructions
 - C. Logic Instructions
 - D. Data Test Instructions
 - E. Index Register and Stack Pointer Instructions
 - F. Branch Instructions
 - G. Condition Code Register Instructions
 - H. Summary of Instruction Set
- VI. New Addressing Modes
 - A. Extended Addressing
 - B. Indexed Addressing
 - C. Instruction Set Summary
- VII. Experiment
- VIII. Unit Examination
- IX. Examination Answers

UNIT 6 THE 6808 MICROPROCESSOR — PART 2

- I. Introduction
- II. Unit Objectives
- III. Unit Activity Guide
- IV. Stack Operations
 - A. Cascade Stack
 - B. Memory Stack
- V. Subroutines
 - A. Jump (JMP) Instruction
 - B. JSR and RTS Instructions
 - C. Nested Subroutines
 - D. Branch to Subroutine (BSR) Instruction
 - E. Summary of Subroutine Instruction
- VI. Input-Output (I/O) Operations
 - A. Output Operations
 - B. Input Operations
 - C. Input-Output Programming
 - D. Program Control of I/O Operations
 - E. Interrupt Control of I/O Operations
- VII. Interrupts
 - A. Reset
 - B. Non-Maskable Interrupts
 - C. Return From Interrupt (RTI) Instruction
 - D. Interrupt Request (IRQ)
 - E. Interrupt Mask Instructions
 - F. Software Interrupt (SWI) Instruction
 - G. Wait for Interrupt (WAI) Instruction
- VIII. Experiment
- IX. Unit Examination
- X. Examination Answers

UNIT 7 PROGRAMMING EXPERIMENTS

- I. Introduction
- II. Experiment 1. Binary/Decimal Training Program
- III. Experiment 2. Hexadecimal/Decimal Training Program
- IV. Experiment 3. Straight Line Programs
- V. Experiment 4. Arithmetic and Logic Instructions
- VI. Experiment 5. Program Branches
- VII. Experiment 6. Additional Instructions
- VIII. Experiment 7. New Addressing Modes
- IX. Experiment 8. Arithmetic Operations
- X. Experiment 9. Stack Operations
- XI. Experiment 10. Subroutines

APPENDIX A DEFINITION OF THE EXECUTABLE INSTRUCTIONS

- I. Nomenclature
- II. Executable Instructions (definition of)
- III. Table A-1. Addressing Formats (1)
- IV. Table A-2. Addressing Formats (2)
- V. Table A-3. Addressing Formats (3)
- VI. Table A-4. Addressing Formats (4)
- VII. Table A-5. Addressing Formats (5)
- VIII. Table A-6. Addressing Formats (6)
- IX. Table A-7. Addressing Formats (7)
- X. Table A-8. Addressing Formats (8)

APPENDIX B DATA SHEETS

- I. MC6800 Data Sheet
- II. MC6801 Data Sheet
- III. MC6802 Data Sheet
- IV. MC6803 Data Sheet
- V. MC6805 Data Sheet
- VI. MC6808 Data Sheet
- VII. MC6809 Data Sheet
- VIII. Positive Power of 2
- IX. Negative Powers of 2
- X. Positive Powers of 8
- XI. Positive Powers of 16.
- XII. Negative Powers of 16.

X



Unit 1
NUMBER SYSTEMS AND CODES

CONTENTS

Introduction	1-3
Unit Objectives	1-4
Unit Activity Guide	1-5
Decimal Number System	1-6
Binary Number System	1-11
Octal Number System	1-20
Hexadecimal Number System	1-29
Binary Codes	1-42
Experiments 1 and 2	1-58
Unit Examination	1-59
Examination Answers	1-61

INTRODUCTION

The purpose of this first unit on microprocessors is to give you a firm foundation in number systems and codes. Binary numbers and codes are the basic language of all microprocessors. Octal and hexadecimal numbers allow easy manipulation of binary numbers and data. Thus, a good foundation in numbers and codes is essential to understanding microprocessors.

This unit will reacquaint you with the decimal number system, then expand the basic concept of numbers to the binary, octal, and hexadecimal systems. Understanding these systems fully will help you understand the many digital codes used with microprocessors. Although this unit can only give you a working knowledge of numbers and codes, you will become more proficient with them as you proceed through the units that follow.

A listing of number system tables has been provided in Appendix B of this course. Appendix B is located at the back of the second binder.

Examine the Unit Objectives listed in the next section to see what you will learn in this unit. Then follow the instructions in the Unit Activity Guide to be sure you perform all of the steps necessary to complete this lesson successfully. Check off each step as you complete it and, in the spaces provided, keep track of the time you spend on each activity.

UNIT OBJECTIVES

When you complete this unit you will have the following knowledge and capabilities:

1. Given any decimal number, you will be able to convert it into its binary, octal, hexadecimal, and BCD equivalent.
2. Given any binary number, you will be able to convert it into its decimal, octal, hexadecimal, and BCD equivalent.
3. Given any octal number, you will be able to convert it into its decimal and binary equivalent.
4. Given any hexadecimal number, you will be able to convert it into its decimal and binary equivalent.
5. Given any BCD code, you will be able to convert it into its decimal and binary equivalent.
6. Given a list of popular digital codes, you will be able to read and identify them including pure binary, natural 8421 BCD, Gray, ASCII, and BAUDOT.
7. You will be able to convert a letter or number into its ASCII binary code, and convert an ASCII binary code into its letter or number equivalent.
8. You will be able to define the following terms:

Radix	BCD
Integer	Gray Code
Decimal	ASCII
Binary	BAUDOT
Octal	Most Significant Bit (MSB)
Hexadecimal	Least Significant Bit (LSB)
Bit	Most Significant Digit (MSD)
Parity	Least Significant Digit (LSD)

UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on the Decimal Number System.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 1 — 6.	_____
<input type="checkbox"/> Read the section on the Binary Number System.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 7 — 13.	_____
<input type="checkbox"/> Read the section on the Octal Number System.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 14 — 19.	_____
<input type="checkbox"/> Read the section on the Hexadecimal Number System.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 20 — 25.	_____
<input type="checkbox"/> Read the section on Binary Codes.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 26 — 36.	_____
<input type="checkbox"/> Perform Experiments 1 and 2.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

DECIMAL NUMBER SYSTEM

The number system we are all familiar with is the decimal number system. This system was originally devised by Hindu mathematicians in India about 400 A.D. The Arabs began to use the system about 800 A.D., where it became known as the Arabic Number System. After it was introduced to the European community about 1200 A.D., the system soon acquired the title “decimal number system.”

A basic distinguishing feature of a number system is its **base** or **radix**. The base indicates the number of characters or digits used to represent quantities in that number system. The decimal number system has a base or radix of 10 because we use the ten digits 0 through 9 to represent quantities. When a number system is used where the base is not known, a subscript is used to show the base. For example, the number 4603_{10} is derived from a number system with a base of 10.

Positional Notation The decimal number system is positional or weighted. This means each digit position in a number carries a particular weight which determines the magnitude of that number. Each position has a weight determined by some power of the number system base, in this case 10. The positional weights are 10^0 (units)*, 10^1 (tens), 10^2 (hundreds), etc. Refer to Figure 1-1 for a condensed listing of powers of 10.

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1,000$
$10^4 = 10,000$
$10^5 = 100,000$
$10^6 = 1,000,000$
$10^7 = 10,000,000$
$10^8 = 100,000,000$
$10^9 = 1,000,000,000$

Figure 1-1
Condensed listing of powers of 10.

*Any number with an exponent of zero is equal to one.

We evaluate the total quantity of a number by considering the specific digits and the weights of their positions. For example, the decimal number 4603 is written in the shorthand notation with which we are all familiar. This number can also be expressed with positional notation.

$$\begin{aligned}(4 \times 10^3) + (6 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) &= \\(4 \times 1000) + (6 \times 100) + (0 \times 10) + (3 \times 1) &= \\4000 + 600 + 0 + 3 &= 4603_{10}\end{aligned}$$

To determine the value of a number, multiply each digit by the weight of its position and add the results.

Fractional Numbers So far, only **integer** or whole numbers have been discussed. An integer is any of the natural numbers, the negatives of these numbers, or zero (that is, 0, 1, 4, 7, etc.). Thus, an integer represents a whole or complete number. But, it is often necessary to express quantities in terms of fractional parts of a whole number.

Decimal fractions are numbers whose positions have weights that are **negative powers of ten** such as $10^{-1} = \frac{1}{10} = 0.1$, $10^{-2} = \frac{1}{100} = 0.01$, etc.

Figure 1-2 provides a condensed listing of negative powers of 10 (decimal fractions).

$$\begin{aligned}10^{-1} &= \frac{1}{10} = 0.1 \\10^{-2} &= \frac{1}{100} = 0.01 \\10^{-3} &= \frac{1}{1000} = 0.001 \\10^{-4} &= \frac{1}{10,000} = 0.0001 \\10^{-5} &= \frac{1}{100,000} = 0.00001 \\10^{-6} &= \frac{1}{1,000,000} = 0.000001\end{aligned}$$

Figure 1-2
Condensed listing of negative
powers of 10.

A radix point (decimal point for base 10 numbers) **separates** the **integer** and **fractional** parts of a number. The integer or whole portion is to the left of the decimal point and has positional weights of units, tens, hundreds, etc. The fractional part of the number is to the right of the decimal point and has positional weights of tenths, hundredths, thousandths, etc. To illustrate this, the decimal number 278.94 can be written with positional notation as shown below.

$$\begin{aligned}(2 \times 10^2) + (7 \times 10^1) + (8 \times 10^0) + (9 \times 10^{-1}) + (4 \times 10^{-2}) &= \\(2 \times 100) + (7 \times 10) + (8 \times 1) + (9 \times 1/10) + (4 \times 1/100) &= \\200 + 70 + 8 + 0.9 + 0.04 &= 278.94_{10}\end{aligned}$$

In this example, the left-most digit (2×10^2) is the **most significant digit** or MSD because it carries the greatest weight in determining the value of the number. The right-most digit, called the **least significant digit** or LSD, has the lowest weight in determining the value of the number. Therefore, as the term implies, the MSD is the digit that will affect the greatest change when its value is altered. The LSD has the smallest effect on the complete number value.

Self-Test Review

- The Radix indicates the number of characters or digits in a number system.
- In the decimal number system, the base or radix is 10.
- Write the following numbers using positional notation.
 - 4563_{10} $(4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (3 \times 10^0)$
 - 26.32_{10} $(2 \times 10^1) + (6 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$
 - 536.9_{10}
- In the decimal number system, the radix point is called the decimal point.
- Convert the following positional notations into their shorthand decimal form.
 - $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2})$ 52.38
 - $(4 \times 10^{-1}) + (6 \times 10^{-2}) + (2 \times 10^{-3})$.462
 - $(3 \times 10^3) + (7 \times 10^2) + (1 \times 10^1) + (0 \times 10^0)$
- The radix point separates the integer and fractional parts of a number.

Answers

1. base or radix.
2. 10.
3. A. $4563_{10} = 4000 + 500 + 60 + 3$
 $= (4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (3 \times 10^0)$
 B. $(2 \times 10^1) + (6 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$
 C. $(5 \times 10^2) + (3 \times 10^1) + (6 \times 10^0) + (9 \times 10^{-1})$
4. decimal point.
5. A. $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2}) =$
 $(5 \times 10) + (2 \times 1) + (3 \times 1/10) + (8 \times 1/100) =$
 $50 + 2 + 0.3 + 0.08 = 52.38_{10}$
 B. 0.462_{10}
 C. 3710_{10}
6. integer or whole, fractional.

$$\begin{array}{cccc} & \downarrow & \downarrow & \\ \frac{1}{1} & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} \end{array}$$

$$\begin{array}{ccc} \frac{1}{1} & \frac{5}{10} & \frac{1}{100} \end{array}$$

$$\begin{array}{ccc} \frac{1}{100} & \frac{1}{100} & \frac{1}{100} \end{array}$$

BINARY NUMBER SYSTEM

The simplest number system that uses positional notation is the binary number system. As the name implies, a **binary** system contains only two elements or states. In a number system this is expressed as a base of 2, using the digits 0 and 1. These two digits have the same basic value as 0 and 1 in the decimal number system.

Because of its simplicity, microprocessors use the binary number system to manipulate data. Binary data is represented by binary digits called **bits**. The term bit is derived from the contraction of **binary digit**. Microprocessors operate on groups of bits which are referred to as words. The binary number 11101101 contains eight bits.

Positional Notation

As with the decimal number system, each bit (digit) position of a binary number carries a particular weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example 2). To evaluate the total quantity of a number, consider the specific bits and the weights of their positions. (Refer to Figure 1-3 for a condensed listing of powers of 2.) For example, the binary number 110101 can be written with positional notation as follows:

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

To determine the decimal value of the binary number 110101, multiply each bit by its positional weight and add the results.

$$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 32 + 16 + 0 + 4 + 0 + 1 = 53_{10}$$

$2^0 = 1_{10}$	$2^6 = 64_{10}$
$2^1 = 2_{10}$	$2^7 = 128_{10}$
$2^2 = 4_{10}$	$2^8 = 256_{10}$
$2^3 = 8_{10}$	$2^9 = 512_{10}$
$2^4 = 16_{10}$	$2^{10} = 1024_{10}$
$2^5 = 32_{10}$	$2^{11} = 2048_{10}$

Figure 1-3
Condensed listing of powers of 2.

Fractional binary numbers are expressed as negative powers of 2. Figure 1-4 provides a condensed listing of negative powers of 2. In positional notation, the binary number 0.1101 can be expressed as follows:

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

To determine the decimal value of the binary number 0.1101, multiply each bit by its positional weight and add the results.

$$(1 \times 1/2) + (1 \times 1/4) + (0 \times 1/8) + (1 \times 1/16) = \\ 0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

In the binary number system, the radix point is called the binary point.

$$2^{-1} = \frac{1}{2} = 0.5_{10}$$

$$2^{-2} = \frac{1}{4} = 0.25_{10}$$

$$2^{-3} = \frac{1}{8} = 0.125_{10}$$

$$2^{-4} = \frac{1}{16} = 0.0625_{10}$$

$$2^{-5} = \frac{1}{32} = 0.03125_{10}$$

$$2^{-6} = \frac{1}{64} = 0.015625_{10}$$

$$2^{-7} = \frac{1}{128} = 0.0078125_{10}$$

$$2^{-8} = \frac{1}{256} = 0.00390625_{10}$$


Figure 1-4
Condensed listing of negative
powers of 2.

Converting Between the Binary and Decimal Number Systems

In working with microprocessors, you will often need to determine the decimal value of binary numbers. In addition, you will find it necessary to convert a specific decimal number into its binary equivalent. The following information shows how such conversions are accomplished.

Binary to Decimal To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where binary 1's occur. The weights of the integer and fractional positions are indicated below.

INTEGER								FRACTIONAL			
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	
128	64	32	16	8	4	2	1	.5	.25	.125	

Binary Point 

As an example, convert the binary number 1010 into its decimal equivalent. Since no binary point is shown, the number is assumed to be an integer number, where the binary point is to the right of the number. The right-most bit, called the **least significant bit** or LSB, has the lowest integer weight of $2^0 = 1$. The left-most bit is the **most significant bit** (MSB) because it carries the greatest weight in determining the value of the number. In this example, it has a weight of $2^3 = 8$. To evaluate the number, add together the weights of the positions where binary 1's appear. In this example, 1's occur in the 2^3 and 2^1 positions. The decimal equivalent is ten.

Binary Number	1	0	1	0					
Position Weights	2^3	2^2	2^1	2^0					
Decimal Equivalent	8	+	0	+	2	+	0	=	10_{10}

To further illustrate this process, convert the binary number 101101.11 into its decimal equivalent.

Binary Number	1	0	1	1	0	1	.1	1									
Position Weights	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}									
Decimal Equivalent																	
	32	+	0	+	8	+	4	+	0	+	1	+	.5	+	.25	=	45.75_{10}

Decimal to Binary A decimal integer number can be converted to a different base or radix through successive divisions by the desired base. To convert a decimal integer number to its binary equivalent, successively divide the number by 2 and note the remainders. When you divide by 2, the remainder will always be 1 or 0.

The remainders form the equivalent binary number.

As an example, the decimal number 25 is converted into its binary equivalent.

$$\begin{array}{rcl}
 25 \div 2 = 12 & \text{with remainder } 1 & \leftarrow \text{LSB} \\
 12 \div 2 = 6 & & 0 \\
 6 \div 2 = 3 & & 0 \\
 3 \div 2 = 1 & & 1 \\
 1 \div 2 = 0 & & 1 \leftarrow \text{MSB}
 \end{array}$$

Divide the decimal number by 2 and note the remainder. Then divide the quotient by 2 and again note the remainder. Then divide the quotient by 2 and again note the remainder. Continue this division process until 0 results. Then collect remainders beginning with the last or most significant bit (MSB) and proceed to the first or least significant bit (LSB). The number $11001_2 = 25_{10}$. Notice that the remainders are collected in the reverse order. That is, the first remainder becomes the least significant bit, while the last remainder becomes the most significant bit.

NOTE: Do not attempt to use a calculator to perform this conversion. It would only supply you with confusing results.

To further illustrate this, the decimal number 175 is converted into its binary equivalent.

$$\begin{array}{rcl}
 175 \div 2 = 87 & \text{with remainder } 1 & \leftarrow \text{LSB} \\
 87 \div 2 = 43 & & 1 \\
 43 \div 2 = 21 & & 1 \\
 21 \div 2 = 10 & & 1 \\
 10 \div 2 = 5 & & 0 \\
 5 \div 2 = 2 & & 1 \\
 2 \div 2 = 1 & & 0 \\
 1 \div 2 = 0 & & 1 \leftarrow \text{MSB}
 \end{array}$$

The division process continues until 0 results. The remainders are collected to produce the number $10101111_2 = 175_{10}$.

To convert a decimal fraction to a different base or radix, multiply the fraction successively by the desired base and record any integers produced by the multiplication as an overflow. For example, to convert the decimal fraction 0.3125 into its binary equivalent, multiply repeatedly by 2.

$0.3125 \times 2 = 0.625 = 0.625$	with overflow	0	← MSB
$0.6250 \times 2 = 1.250 = 0.250$		1	
$0.2500 \times 2 = 0.500 = 0.500$		0	
$0.5000 \times 2 = 1.000 = 0$		1	← LSB

These multiplications will result in numbers with a 1 or 0 in the units position (the position to the left of the decimal point). By recording the value of the units position, you can construct the equivalent binary fraction. This units position value is called the "overflow." Therefore, when 0.3125 is multiplied by 2, the overflow is 0. This becomes the most significant bit (MSB) of the binary equivalent fraction. Then 0.625 is multiplied by 2. Since the product is 1.25, the overflow is 1. When there is an overflow of 1, it is effectively subtracted from the product when the value is recorded. Therefore, only 0.25 is multiplied by 2 in the next multiplication process. This method continues until an overflow with no fraction results. It is important to note that you can not always obtain 0 when you multiply by 2. Therefore, you should only continue the conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (binary) point with the MSB and proceed to the LSB. This is the same order in which the overflows were produced. The number $0.0101_2 = 0.3125_{10}$.

To further illustrate this process, the decimal fraction 0.90625 is converted into its binary equivalent.

$0.90625 \times 2 = 1.8125 = 0.8125$	with overflow	1	← MSB
$0.81250 \times 2 = 1.6250 = 0.6250$		1	
$0.62500 \times 2 = 1.2500 = 0.2500$		1	
$0.25000 \times 2 = 0.5000 = 0.5000$		0	
$0.50000 \times 2 = 1.0000 = 0$		1	← LSB

The multiplication process continues until either 0 or the desired precision is obtained. The overflows are then collected beginning with the MSB at the binary (radix) point and proceeding to the LSB. The number $0.11101_2 = 0.90625_{10}$.

25	12 R 1
25	6 R 0
	3 R 0
	1 R 1
	0
	10010
	16
2	87 R 1
175	43 R 1
174	21 R 1
	10 R 1
	5 R 0
	2 R 1
	1 R 0
	0 R 1
1010111	1
76843210	2
	4
	8
	16
	32
	64
	128
	256

If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each number portion. After you convert the binary integer and binary fraction, recombine them. For example, the decimal number 14.375 is converted into its binary equivalent.

$$14.375_{10} = 14_{10} + 0.375_{10}$$

$$14 \div 2 = 7$$

$$7 \div 2 = 3$$

$$3 \div 2 = 1$$

$$1 \div 2 = 0$$

with remainder 0 ← LSB

1

1

1 ← MSB

$$\boxed{14_{10} = 1110_2}$$

$$0.375 \times 2 = 0.75 = 0.75$$

with overflow 0 ← MSB

$$0.750 \times 2 = 1.50 = 0.50$$

1

$$0.500 \times 2 = 1.00 = 0$$

1 ← LSB

$$\boxed{0.375_{10} = 0.011_2}$$

$$14.375_{10} = 14_{10} + 0.375_{10} = 1110_2 + 0.011_2 = 1110.011_2$$

Self-Test Review

7. The base or radix of the binary number system is 2.

8. A binary digit is called a bit.

9. Convert the following binary integers to decimal.

- A. 101101
B. 1001
C. 1101100

10. Convert the following binary fractions to decimal.

- A. 0.011
B. 0.01101
C. 0.1001

11. Convert the following decimal integers to binary.

- A. 63
B. 12
C. 132

12. Convert the following decimal fractions to binary.

- A. 0.4375
B. 0.96875
C. 0.625

13. Convert 13.125_{10} to binary.

11011001

Answers

7. 2
8. bit
9. A. $101101_2 =$
 $(1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$
 $32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$
 B. $1001_2 = 9_{10}$
 C. $1101100_2 = 108_{10}$
10. A. $0.011_2 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) =$
 $0 + \frac{1}{4} + \frac{1}{8} = 0 + 0.25 + 0.125 = 0.375_{10}$
 B. $0.01101_2 = 0.40625_{10}$
 C. $0.1001_2 = 0.5625_{10}$
11. A. $63 \div 2 = 31$ with remainder 1 ← LSB
 $31 \div 2 = 15$ 1
 $15 \div 2 = 7$ 1
 $7 \div 2 = 3$ 1
 $3 \div 2 = 1$ 1
 $1 \div 2 = 0$ 1 ← MSB
 $63_{10} = 111111_2$
- B. $12_{10} = 1100_2$
 C. $132_{10} = 10000100_2$

12. A. $0.4375 \times 2 = 0.875 = 0.875$ with overflow 0 \leftarrow MSB
 $0.8750 \times 2 = 1.750 = 0.750$ 1
 $0.7500 \times 2 = 1.500 = 0.500$ 1
 $0.5000 \times 2 = 1.000 = 0$ 1 \leftarrow LSB

$$\boxed{0.4375_{10} = 0.0111_2}$$

B. $0.96875_{10} = 0.11111_2$

C. $0.625_{10} = 0.101_2$

13. $13.125_{10} = 13_{10} + 0.125_{10}$

$$\begin{array}{ll} 13 \div 2 = 6 & \text{with remainder } 1 \leftarrow \text{LSB} \\ 6 \div 2 = 3 & 0 \\ 3 \div 2 = 1 & 1 \\ 1 \div 2 = 0 & 1 \leftarrow \text{MSB} \end{array}$$

$$\boxed{13_{10} = 1101_2}$$

$$\begin{array}{ll} 0.125 \times 2 = 0.25 = 0.25 & \text{with overflow } 0 \leftarrow \text{MSB} \\ 0.250 \times 2 = 0.50 = 0.50 & 0 \\ 0.500 \times 2 = 1.00 = 0 & 1 \leftarrow \text{LSB} \end{array}$$

$$\boxed{0.125_{10} = 0.001_2}$$

$$13.125_{10} = 13_{10} + 0.125_{10} = 1101_2 + 0.001_2 = 1101.001_2$$

OCTAL NUMBER SYSTEM

Octal is another number system that is often used with microprocessors. It has a base (radix) of 8, and uses the digits 0 through 7. These eight digits have the same basic value as the digits 0—7 in the decimal number system.

As with the binary number system, each digit position of an octal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example, 8). To evaluate the total quantity of a number, consider the specific digits and the weights of their positions. Refer to Figure 1-5 for a condensed listing of powers of 8. For example, the octal number 372.01 can be written with positional notation as follows:

$$(3 \times 8^2) + (7 \times 8^1) + (2 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$$

The decimal value of the octal number 372.01 is determined by multiplying each digit by its positional weight and adding the results. As with decimal and binary numbers, the radix (octal) point separates the integer from the fractional part of the number.

$$(3 \times 64) + (7 \times 8) + (2 \times 1) + (0 \times 0.125) + (1 \times 0.015625) = 192 + 56 + 2 + 0 + 0.015625 = 250.015625_{10}$$

$$8^{-4} = \frac{1}{4096} = 0.000244140625_{10}$$

$$8^{-3} = \frac{1}{512} = 0.001953125_{10}$$

$$8^{-2} = \frac{1}{64} = 0.015625_{10}$$

$$8^{-1} = \frac{1}{8} = 0.125_{10}$$

$$1_{10} = 8^0$$

$$8_{10} = 8^1$$

$$64_{10} = 8^2$$

$$512_{10} = 8^3$$

$$4096_{10} = 8^4$$

$$32768_{10} = 8^5$$

$$262144_{10} = 8^6$$

Figure 1-5

Condensed listing of powers of 8.

Conversion From Decimal to Octal

Decimal to octal conversion is accomplished in the same manner as decimal to binary, with one exception; the base number is now 8 rather than 2. As an example, the decimal number 194 is converted into its octal equivalent.

$$\begin{array}{rcl}
 194 \div 8 = 24 \text{ with remainder } 2 & \leftarrow & \text{LSD} \\
 24 \div 8 = 3 & & 0 \\
 3 \div 8 = 0 & & 3 \leftarrow \text{MSD}
 \end{array}$$

Divide the decimal number by 8 and note the remainder. (The remainder can be any number from 0 to 7.)

Then divide the quotient by 8 and again note the remainder. Continue dividing until 0 results. Finally, collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number $302_8 = 194_{10}$. Figure 1-6 illustrates the relationship between the first several decimal, octal, and binary integers.

DECIMAL	OCTAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000
17	21	10001
18	22	10010
19	23	10011
20	24	10100

Figure 1-6
Sample comparison of decimal,
octal, and binary integers.

To further illustrate this process, the decimal number 175 is converted into its octal equivalent.

$$\begin{array}{rcl} 175 \div 8 = 21 \text{ with remainder } 7 & \leftarrow & \text{LSD} \\ 21 \div 8 = 2 & & 5 \\ 2 \div 8 = 0 & & 2 \leftarrow \text{MSD} \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number $257_8 = 175_{10}$.

To convert a decimal fraction to an octal fraction, multiply the fraction successively by 8 (octal base). As an example, the decimal fraction 0.46875 is converted into its octal equivalent.

$$\begin{array}{rcl} 0.46875 \times 8 = 3.75 = 0.75 \text{ with overflow } 3 & \leftarrow & \text{MSD} \\ 0.75000 \times 8 = 6.00 = 0 & & 6 \leftarrow \text{LSD} \end{array}$$

Multiply the decimal number by 8. If the product exceeds one, subtract the integer (overflow) from the product. Then multiply the product fraction by 8 and again note any "overflow." Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 8. Therefore, you should only continue this conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (octal point) with the MSD and proceed to the LSD. The number $0.36_8 = 0.46875_{10}$. Figure 1-7 illustrates the relationship between decimal, octal, and binary fractions.

Now, the decimal fraction 0.136 will be converted into its octal equivalent with four-place precision.

$$\begin{array}{rcl} 0.136 \times 8 = 1.088 = 0.088 \text{ with overflow } 1 & \leftarrow & \text{MSD} \\ 0.088 \times 8 = 0.704 = 0.704 & & 0 \\ 0.704 \times 8 = 5.632 = 0.632 & & 5 \\ 0.632 \times 8 = 5.056 = 0.056 & & 5 \leftarrow \text{LSD} \\ 0.136_{10} \approx 0.1055_8 \end{array}$$

The number 0.1055_8 approximately equals 0.136_{10} . If you convert 0.1055_8 back to decimal (using positional notation), you will find $0.1055_8 = 0.135986328125_{10}$. This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

DECIMAL	OCTAL	BINARY
0.015625	0.01	0.000001
0.03125	0.02	0.00001
0.046875	0.03	0.000011
0.0625	0.04	0.0001
0.078125	0.05	0.000101
0.09375	0.06	0.00011
0.109375	0.07	0.000111
0.125	0.1	0.001
0.140625	0.11	0.001001
0.15625	0.12	0.00101
0.171875	0.13	0.001011
0.1875	0.14	0.0011
0.203125	0.15	0.001101
0.21875	0.16	0.00111
0.234375	0.17	0.001111
0.25	0.2	0.01
0.265625	0.21	0.010001
0.28125	0.22	0.01001
0.296875	0.23	0.010011
0.3125	0.24	0.0101

Figure 1-7

Sample comparison of decimal, octal, and binary fractions.

As with decimal to binary conversion of a number that contains both an integer and fraction, decimal to octal conversion requires two operations. You must separate the integer from the fraction, then perform the appropriate conversion on each number. After you convert them, you must recombine the octal integer and octal fraction. For example, convert the decimal number 124.78125 into its octal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 8 = 15 \quad \text{with remainder} \quad 4 \quad \leftarrow \text{LSD}$$

$$15 \div 8 = 1 \quad 7$$

$$1 \div 8 = 0 \quad 1 \quad \leftarrow \text{MSD}$$

$$124_{10} = 174_8$$

$$0.78125 \times 8 = 6.25 = 0.25 \quad \text{with overflow} \quad 6 \quad \leftarrow \text{MSD}$$

$$0.25000 \times 8 = 2.00 = 0 \quad 2 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.62_8$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 174_8 + 0.62_8 = 174.62_8$$

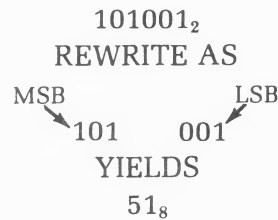
Converting Between the Octal and Binary Number Systems

Microprocessors manipulate data using the binary number system. However, when larger quantities are involved, the binary number system can become cumbersome. Therefore, other number systems are frequently used as a form of binary shorthand to speed-up and simplify data entry and display. The octal number system is one of the systems that is used in this manner. It is similar to the decimal number system, which makes it easier to understand numerical values. In addition, conversion between binary and octal is readily accomplished because of the value structure of octal. Figures 1-6 and 1-7 illustrate the relationship between octal and binary integers and fractions.

As you know, three bits of a binary number exactly equal eight value combinations. Therefore, you can represent a 3-bit binary number with a 1-digit octal number.

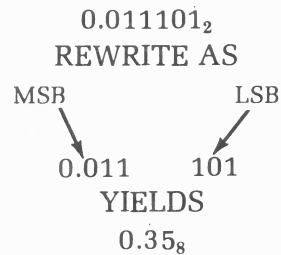
$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4 + 0 + 1 = 5_8$$

Because of this relationship, converting binary to octal is simple and straight forward. For example, binary number 101001 is converted into its octal equivalent.



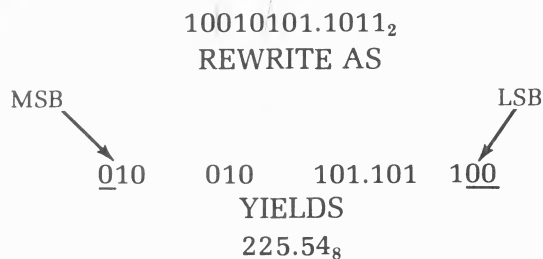
To convert a binary number to octal, first separate the number into groups containing three bits, beginning with the least significant bit. Then convert each 3-bit group into its octal equivalent. This gives you an octal number equal in value to the binary number.

Binary fractions can also be converted to their octal equivalents using the same process, with one exception. The binary bits must be separated into groups of three beginning with the most significant bit. For example, the binary fraction 0.011101_2 is converted into its octal equivalent.



Again, you must first separate the binary number into groups of three beginning at the radix (binary) point. Then convert each 3-bit group into its octal equivalent.

To separate binary numbers into 3-bit groups when the number does not contain the necessary bits, add zeros to the number until the number can be separated into 3-bit groups. For example, binary number 10010101.1011_2 is converted into its octal equivalent.



As before, the integer part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the third group contains only two bits. However, a zero can be added to the group without changing the value of the binary number. Next, the fractional part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the second group contains only one bit. By adding two zeros to the group, the group is complete with no change in the value of the binary number.

NOTE: Whenever you add zeros to a **binary integer**, always place them to the **left** of the most significant bit. When you add zeros to a **binary fraction**, always place them to the **right** of the least significant bit.

1101110.01₂
REWRITE AS
MSBLSB
001101110.010
YIELDS
156.2₈

Converting octal to binary is just the opposite of the previous process. You simply convert each octal number into its 3-bit binary equivalent. For example, convert the octal number 75.3 into its binary equivalent.

75.3₈
YIELDS
MSB ↘ 111 101.011 ↙ LSB
REWRITE AS
111101.011₂

The above example is a simple conversion. Now a more complex octal number (1752.714) will be converted to a binary number.

1752.714₈
YIELDS

MSB ↘ 001 111 101 010.111 001 100 ↙ LSB

REWRITE AS
1111101010.1110011₂

Again, each octal digit is converted into its 3-bit binary equivalent. However, in this example, there are two insignificant zeros in front of the MSB and after the LSB. Since these zeros have no value, they should be removed from the final result.

Self-Test Review

14. The base or radix of the octal number system is 8.
15. Convert the following decimal integers to octal.
- A. 156 234
- B. 32 40
- C. 1785 323
16. Convert the following decimal fractions to octal. Do not use greater than 4-place precision.
- A. 0.1432 0.23
- B. 0.8125 0.65
- C. 0.6832 0.55
17. Convert 735.984375_{10} to octal.
18. Convert the following binary numbers to octal.
- A. 10000111.01101 207.32
- B. 11101.0101 35.24
- C. 1001101.000001 11.0001
19. Convert the following octal numbers to binary.
- A. 372.61 111011010.1101
- B. 11.001 101.001
- C. 3251.034 110110101.0011

Answers

14. 8

15. A. $156 \div 8 = 19$ with remainder 4 ← LSD
 $19 \div 8 = 2$ 3
 $2 \div 8 = 0$ 2 ← MSD

$$\boxed{156_{10} = 234_8}$$

B. $32_{10} = 40_8$

C. $1785_{10} = 3371_8$

16. A. $0.1432 \times 8 = 1.1456 = 0.1456$ overflow 1 ← MSD
 $0.1456 \times 8 = 1.1648 = 0.1648$ 1
 $0.1648 \times 8 = 1.3184 = 0.3184$ 1
 $0.3184 \times 8 = 2.5472 = 0.5472$ 2 ← LSD

$$\boxed{0.1432_{10} = 0.1112_8}$$

B. $0.8125_{10} = 0.64_8$

C. $0.6832_{10} = 0.5356_8$

17. $735.984375_{10} = 735_{10} + 0.984375_{10}$
 $= 735 \div 8 = 91$ with remainder 7 ← LSD
 $91 \div 8 = 11$ 3
 $11 \div 8 = 1$ 3
 $1 \div 8 = 0$ 1 ← MSD

$$\boxed{735_{10} = 1337_8}$$

$0.984375 \times 8 = 7.875 = 0.875$ overflow
 7 ← MSD

$0.875000 \times 8 = 7.00 = 0$ 7 ← LSD

$$\boxed{0.984375_{10} = 0.77_8}$$

$$735.984375_{10} = 735_{10} + 0.984375_{10} = 1337_8 + 0.77_8 = 1337.77_8$$

18. A. $10000111.01101_2 = 010\ 000\ 111.011\ 010_2$
 $= 207.32_8$

B. $11101.0101_2 = 35.24_8$

C. $1001101.000001_2 = 115.01_8$

19. A. $372.61_8 = 011\ 111\ 010.110\ 001_2 = 11111010.110001_2$.

B. $11.001_8 = 1001.000000001_2$

C. $3251.034_8 = 11010101001.0000111_2$

HEXADECIMAL NUMBER SYSTEM

Hexadecimal is another number system that is often used with microprocessors. It is similar in value structure to the octal number system, and thus allows easy conversion with the binary number system. Because of this feature and the fact that hexadecimal simplifies data entry and display to a greater degree than octal, you will use hexadecimal more often than any other number system in this course. As the name implies, hexadecimal has a base (radix) of 16_{10} . It uses the digits 0 through 9 and the letters A through F.

The letters are used because it is necessary to represent 16_{10} different values with a single digit for each value. Therefore, the letters A through F are used to represent the number values 10_{10} through 15_{10} . The following discussion will compare the decimal number system with the hexadecimal number system.

All of the numbers are of equal value between systems ($0_{10} = 0_{16}$, $3_{10} = 3_{16}$, $9_{10} = 9_{16}$, etc.). For numbers greater than 9, this relationship exists: $10_{10} = A_{16}$, $11_{10} = B_{16}$, $12_{10} = C_{16}$, $13_{10} = D_{16}$, $14_{10} = E_{16}$, and $15_{10} = F_{16}$. Using letters in counting may appear awkward until you become familiar with the system. Figure 1-8 illustrates the relationship between decimal and hexadecimal integers, while Figure 1-9 illustrates the relationship between decimal and hexadecimal fractions.

DECIMAL	HEXADECIMAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100
21	15	10101
22	16	10110
23	17	10111
24	18	11000
25	19	11001
26	1A	11010
27	1B	11011
28	1C	11100
29	1D	11101
30	1E	11110
31	1F	11111
32	20	100000
33	21	100001
34	22	100010
35	23	100011

Figure 1-8
Sample comparison of decimal,
hexadecimal, and binary integers.

DECIMAL	HEXADECIMAL	BINARY
0.00390625	0.01	0.00000001
0.0078125	0.02	0.0000001
0.01171875	0.03	0.00000011
0.015625	0.04	0.000001
0.01953125	0.05	0.00000101
0.0234375	0.06	0.0000011
0.02734375	0.07	0.00000111
0.03125	0.08	0.00001
0.03515625	0.09	0.00001001
0.0390625	0.0A	0.0000101
0.04296875	0.0B	0.00001011
0.046875	0.0C	0.00011
0.05078125	0.0D	0.00001101
0.0546875	0.0E	0.0000111
0.05859375	0.0F	0.00001111
0.0625	0.1	0.0001
0.06640625	0.11	0.00010001
0.0703125	0.12	0.0001001
0.07421875	0.13	0.00010011
0.078125	0.14	0.000101
0.08203125	0.15	0.00010101
0.0859375	0.16	0.0001011
0.08984375	0.17	0.00010111
0.09375	0.18	0.00011
0.09765625	0.19	0.00011001
0.1015625	0.1A	0.0001101
0.10546875	0.1B	0.00011011
0.109375	0.1C	0.000111
0.11328125	0.1D	0.00011101
0.1171875	0.1E	0.0001111
0.12109375	0.1F	0.00011111
0.125	0.2	0.001

Figure 1-9
Sample comparison of decimal,
hexadecimal, and binary fractions.

$$16^{-1} = \frac{1}{16}$$

$$16^{-2} = \frac{1}{16^2}$$

$$16^3$$

$$10A = 0 \times \frac{1}{16} + 1 \times \frac{1}{16}$$

As with the previous number systems, each digit position of a hexadecimal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example, 16_{10}). The total quantity of a number can be evaluated by considering the specific digits and the weights of their positions. (Refer to Figure 1-10 for a condensed listing of powers of 16_{10} .) For example, the hexadecimal number E5D7.A3 can be written with positional notation as follows:

$$(E \times 16^3) + (5 \times 16^2) + (D \times 16^1) + (7 \times 16^0) + (A \times 16^{-1}) + (3 \times 16^{-2})$$

The decimal value of the hexadecimal number E5D7.A3 is determined by multiplying each digit by its positional weight and adding the results. As with the previous number systems, the radix (hexadecimal) point separates the integer from the fractional part of the number.

$$\begin{aligned} &(14 \times 4096) + (5 \times 256) + (13 \times 16) + (7 \times 1) + (10 \times 1/16) + (3 \times 1/256) = \\ &57344 + 1280 + 208 + 7 + 0.625 + 0.01171875 = \\ &58839.63671875_{10} \end{aligned}$$

$$\begin{aligned} 16^{-4} &= \frac{1}{65536} = 0.0000152587890625_{10} \\ 16^{-3} &= \frac{1}{4096} = 0.000244140625_{10} \\ 16^{-2} &= \frac{1}{256} = 0.00390625_{10} \\ 16^{-1} &= \frac{1}{16} = 0.0625_{10} \end{aligned}$$

$$\begin{aligned} 1_{10} &= 16^0 \\ 16_{10} &= 16^1 \\ 256_{10} &= 16^2 \\ 4096_{10} &= 16^3 \\ 65536_{10} &= 16^4 \\ 1048576_{10} &= 16^5 \\ 16777216_{10} &= 16^6 \end{aligned}$$

Figure 1-10
Condensed listing of powers of 16.

Conversion From Decimal to Hexadecimal

Decimal to hexadecimal conversion is accomplished in the same manner as decimal to binary or octal, but with a base number of 16_{10} . As an example, the decimal number 156 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 156 \div 16 = 9 & \text{with remainder } 12 = C & \leftarrow & \text{LSD} \\ 9 \div 16 = 0 & & 9 = 9 & \leftarrow \text{MSD} \end{array}$$

Divide the decimal number by 16_{10} and note the remainder. If the remainder exceeds 9, convert the 2-digit number to its hexadecimal equivalent ($12_{10} = C$ in this example). Then divide the quotient by 16 and again note the remainder. Continue dividing until a quotient of 0 results. Then collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number $9C_{16} = 156_{10}$. NOTE: The letter H after a number is sometimes used to indicate hexadecimal. However, this course will always use the subscript 16.

To further illustrate this, the decimal number 47632 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 47632 \div 16 = 2977 & \text{with remainder } 0 = 0 & \leftarrow & \text{LSD} \\ 2977 \div 16 = 186 & & 1 = 1 & \\ 186 \div 16 = 11 & & 10 = A & \\ 11 \div 16 = 0 & & 11 = B & \leftarrow \text{MSD} \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number $BA10_{16} = 47632_{10}$. Remember, any remainder that exceeds the digit 9 must be converted to its letter equivalent. (In this example, $10 = A$, and $11 = B$.)

To convert a decimal fraction to a hexadecimal fraction, multiply the fraction successively by 16_{10} (hexadecimal base). As an example the decimal fraction 0.78125 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 0.78125 \times 16 = 12.5 = 0.5 & \text{with overflow} & 12 = C & \leftarrow \text{MSD} \\ 0.50000 \times 16 = 8.0 = 0 & & 8 = 8 & \leftarrow \text{LSD} \end{array}$$

Multiply the decimal by 16_{10} . If the product exceeds one, subtract the integer (overflow) from the product. If the "overflow" exceeds 9, convert the 2-digit number to its hexadecimal equivalent. Then multiply the product fraction by 16_{10} and again note any overflow. Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 16. Therefore, you should only continue the conversion to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix point with the MSD and proceed to the LSD. The number $0.C8_{16} = 0.78125_{10}$.

Now the decimal fraction 0.136 will be converted into its hexadecimal equivalent with five-place precision.

$0.136 \times 16 = 2.176 = 0.176$	overflow	$2 = 2 \rightarrow$	MSD
$0.176 \times 16 = 2.816 = 0.816$		$2 = 2$	
$0.816 \times 16 = 13.056 = 0.056$		$13 = D$	
$0.056 \times 16 = 0.896 = 0.896$		$0 = 0$	
$0.896 \times 16 = 14.336 = 0.336$		$14 = E \rightarrow$	LSD

The number $0.22D0E_{16}$ approximately equals 0.136_{10} . If you convert $0.22D0E_{16}$ back to decimal (using positional notation), you will find $0.22D0E_{16} = 0.1359996795654296875_{16}$. This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

As shown in this section, conversion of an integer from decimal to hexadecimal requires a different technique than for conversion of a fraction. Therefore, when you convert a hexadecimal number composed of an integer and a fraction, you must separate the integer and fraction, then perform the appropriate operation on each. After you convert them, you must recombine the integer and fraction. For example, the decimal number 124.78125 is converted into its hexadecimal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 16 = 7 \quad \text{with remainder } 12 = C \quad \leftarrow \text{LSD}$$

$$7 \div 16 = 0 \quad \quad \quad 7 = 7 \quad \leftarrow \text{MSD}$$

$$124_{10} = 7C_{16}$$

$$0.78125 \times 16 = 12.5 = 0.5 \quad \text{overflow} \quad 12 = C \quad \leftarrow \text{MSD}$$

$$0.50000 \times 16 = 8.0 = 0 \quad \quad \quad 8 = 8 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.C8_{16}$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 7C_{16} + 0.C8_{16} = 7C.C8_{16}$$

First separate the decimal integer and fraction. Then convert the integer and fraction to hexadecimal.

Finally, recombine the integer and fraction.

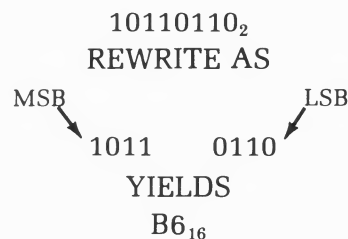
Converting Between the Hexadecimal and Binary Number Systems

Previously, the octal number system was described as an excellent shorthand form to express large binary quantities. This method is very useful with many microprocessors. The trainer used with this course uses the hexadecimal number system to represent binary quantities. As a result, frequent conversions from binary-to-hexadecimal are necessary. Figures 1-8 and 1-9 illustrate the relationship between hexadecimal and binary integers and fractions.

As you know, four bits of a binary number exactly equal 16_{10} value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number:

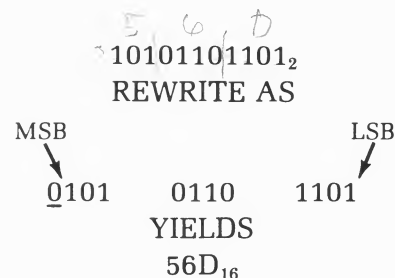
$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13_{10} = D_{16}$$

Because of this relationship, converting binary to hexadecimal is simple and straightforward. For example, binary number 10110110 is converted into its hexadecimal equivalent.



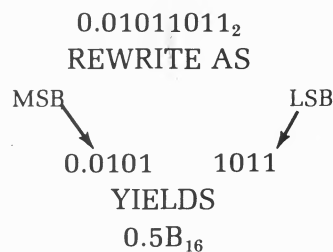
To convert a binary number to hexadecimal, first separate the number into groups containing four bits, beginning with the least significant bit. Then convert each 4-bit group into its hexadecimal equivalent. Don't forget to use letter digits as required. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary number (10101101101) into its hexadecimal equivalent.



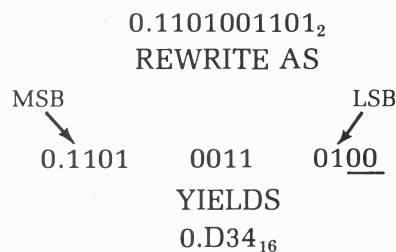
Again, the binary number is separated into 4-bit groups beginning with the LSB. However, the third group contains only three bits. Since each group must contain four bits, a zero must be added after the MSB. The third group will then have four bits with no change in the value of the binary number. Now each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary integer, always place them to the left of the most significant bit.**

Binary fractions can also be converted to their hexadecimal equivalents using the same process, with one exception; the binary bits are separated into groups of four, beginning with the most significant bit (at the radix point). For example, the binary fraction 0.01011011 is converted into its hexadecimal equivalent.



Again, you must separate the binary number into groups of four, beginning with the radix point. Then convert each 4-bit group into its hexadecimal equivalent. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary fraction (0.1101001101) into its hexadecimal equivalent.



Separate the binary number into 4-bit groups, beginning at the radix (binary) point (MSB). Note that the third group contains only two bits. Since each group must contain four bits, two zeros must be added after the LSB. The third group will then have four bits with no change in the value of the binary number. Now, each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary fraction, always place them to the right of the least significant bit.**

Now, a binary number containing both an integer and a fraction (110110101.01110111) will be converted into its hexadecimal equivalent.

110110101.01110111₂
 REWRITE AS
 MSB ↙ ↘ LSB
0001 1011 0101.0111 0111
 YIELDS
 1B5.77₁₆

The integer part of the number is separated into groups of four, **beginning** at the radix point. Note that three zeros were added to the third group to complete the group. The fractional part of the number is separated into groups of four, **beginning** at the radix point. (No zeros were needed to complete the fractional groups.) The integer and fractional 4-bit groups are then converted to hexadecimal. The number $110110101.01110111_2 = 1B5.77_{16}$. **Never** shift the radix point in order to form 4-bit groups.

Converting hexadecimal to binary is just the opposite of the previous process; simply convert each hexadecimal number into its 4-bit binary equivalent. For example, convert the hexadecimal number 8F.41 into its binary equivalent.

8F.41₁₆
YIELDS

MSB ↘ 1000 1111.0100 0001 ↙ LSB

REWRITE AS
10001111.01000001₂

Convert each hexadecimal digit into a 4-bit binary number. Then condense the 4-bit groups to form the binary value equal to the hexadecimal value. The number $8F.41_{16} = 10001111.01000001_2$.

Now, the hexadecimal number $175.4E_{16}$ will be converted into its binary equivalent.

$175.4E_{16}$
 YIELDS
 MSB LSB
 $\underline{0001\ 0111\ 0101.0100\ 1110}$
 REWRITE AS
 101110101.0100111_2

6
16) 96
6

Again, each hexadecimal digit is converted into its 4-bit binary equivalent. However, in this example there are three insignificant zeros in front of the MSB and one after the LSB. Since these zeros have no value, they should be removed from the final result.

78
16) 1578
144
133
128
5

Self-Test Review

20. The base or radix of the hexadecimal number system is 16.

21. Convert the following decimal integers to hexadecimal.

- A. 783
B. 5372
C. 957

30F
14FC
3BD

29
16) 957
80
157
144
13

22. Convert the following decimal fractions to hexadecimal. Do not use greater than four-place precision.

- A. 0.653
B. 0.109375
C. 0.4567

A72B
19
74EA

48
16) 723
64
143
128
15

23. Convert 1573.125_{10} to its hexadecimal equivalent. 625.2

24. Convert the following binary numbers to hexadecimal.

- A. 100001101.01011
B. 11111011001.01
C. 110001101.00010010101

100.58
709.4
18D.12A

375
16) 5372
48
57

25. Convert the following hexadecimal numbers to binary.

- A. AE7.D2
B. 2C5.21F8
C. 1B6.64E

101011100111.1101001
1011000101.001000011111
110110110.0110100111

48
16) 5372
48
57
48
92
82
10

Answers

20. 16_{10} .

21. A. $783 \div 16 = 48$ with remainder $15 = F \leftarrow \text{LSD}$
 $48 \div 16 = 3$ $0 = 0$
 $3 \div 16 = 0$ $3 = 3 \leftarrow \text{MSD}$

$$783_{10} = 30F_{16}$$

B. $5372_{10} = 14FC_{16}$

C. $957_{10} = 3BD_{16}$

22. A. $0.653 \times 16 = 10.448 = 0.448$ with overflow $10 = A \leftarrow \text{MSD}$
 $0.448 \times 16 = 7.168 = 0.168$ $7 = 7$
 $0.168 \times 16 = 2.688 = 0.688$ $2 = 2$
 $0.688 \times 16 = 11.008 = 0.008$ $11 = B \leftarrow \text{LSD}$

$$0.653_{10} = 0.A72B_{16}$$

B. $0.109375_{10} = 0.1C_{16}$

C. $0.4567_{10} = 0.74EA_{16}$

23. A. $1573.125_{10} = 1573_{10} + 0.125_{10}$
 $1573 \div 16 = 98$ with remainder $5 = 5 \leftarrow \text{LSD}$
 $98 \div 16 = 6$ $2 = 2$
 $6 \div 16 = 0$ $6 = 6 \leftarrow \text{MSD}$

$$1573_{10} = 625_{16}$$

$0.125 \times 16 = 2.00 = 0$ with overflow $2 = 2 \leftarrow \text{MSD}$
 $\leftarrow \text{LSD}$

$$0.125_{10} = 0.2_{16}$$

$$1573.125_{10} = 1573_{10} + 0.125_{10} = 625_{16} + 0.2_{16} = 625.2_{16}$$

24. A. $100001101.01011_2 = 0001\ 0000\ 1101.0101\ 1000_2$
 $= 10D.58_{16}$

B. $11111011001.01_2 = 7D9.4_{16}$

C. $110001101.00010010101_2 = 18D.12A_{16}$

25. A. $AE7.D2_{16} = 1010\ 1110\ 0111.1101\ 0010_2$
 $= 101011100111.1101001_2$

B. $2C5.21F8_{16} = 1011000101.001000011111_2$

C. $1B6.64E_{16} = 110110110.01100100111_2$

BINARY CODES

Converting a decimal number into its binary equivalent is called “coding.” A decimal number is expressed as a binary code or binary number. The **binary number system**, as discussed, is known as the pure binary code. This name distinguishes it from other types of binary codes. This section will discuss some of the other types of binary codes used in computers.

Binary Coded Decimal

The decimal number system is easy to use because it is so familiar. The binary number system is less convenient to use because it is less familiar. It is difficult to quickly glance at a binary number and recognize its decimal equivalent. For example, the binary number 1010011 represents the decimal number 83. It is difficult to tell immediately by looking at the number what its decimal value is. However, within a few minutes, using the procedures described earlier, you could readily calculate its decimal value. The amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Engineers recognized this problem early and developed a special form of binary code that was more compatible with the decimal system. Because so many digital devices, instruments and equipment use decimal input and output, this special code has become very widely used and accepted. This special compromise code is known as binary coded decimal (BCD). The BCD code combines some of the characteristics of both the binary and decimal number systems.

8421 BCD Code The BCD code is a system of representing the decimal digits 0 through 9 with a four-bit binary code. This BCD code uses the standard 8421 position **weighting system** of the pure binary code. The standard 8421 BCD code and the decimal equivalents are shown in Figure 1-11, along with a special Gray code that will be described later. As with the pure binary code, you can convert the BCD numbers into their decimal equivalents by simply adding together the weights of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system.

DECIMAL	8421 BCD	GRAY	BINARY
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	0101
6	0110	0101	0110
7	0111	0100	0111
8	1000	1100	1000
9	1001	1101	1001
10	0001 0000	1111	1010
11	0001 0001	1110	1011
12	0001 0010	1010	1100
13	0001 0011	1011	1101
14	0001 0100	1001	1110
15	0001 0101	1000	1111

Figure 1-11
Codes.

To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit. For example, the decimal integer 834 in BCD would be 1000 0011 0100. Each decimal digit is represented by its equivalent 8421 4-bit code. A space is left between each 4-bit group to avoid confusing the BCD format with the pure binary code. This method of representation also applies to decimal fractions. For example, the decimal fraction 0.764 would be 0.0111 0110 0100 in BCD. Again, each decimal digit is represented by its equivalent 8421 4-bit code, with a space between each group.

An advantage of the BCD code is that the ten BCD code combinations are easy to remember. Once you begin to work with binary numbers regularly, the BCD numbers may come to you as quickly and automatically as decimal numbers. For that reason, by simply glancing at the BCD representation of a decimal number you can make the conversion almost as quickly as if it were already in decimal form. As an example, convert a BCD number into its decimal equivalent.

$$0110\ 0010\ 1000.1001\ 0101\ 0100 = 628.954_{10}$$

The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does with pure binary notation. For example, the decimal number 83 in pure binary form is 1010011. In BCD code the decimal number 83 is written as 1000 0011. In the pure binary code, it takes only seven bits to represent the number 83. In BCD form, it takes eight bits. It is inefficient because, for each bit in a data word, there is usually some digital circuitry associated with it. The extra circuitry associated with the BCD code costs more, increases equipment complexity, and consumes more power. Arithmetic operations with BCD numbers are also more time consuming and complex than those with pure binary numbers. With four bits of binary information, you can represent a total of $2^4 = 16$ different states or the decimal number equivalents 0 through 15. In the BCD system, six of these states (10-15), are wasted. When the BCD number system is used, some efficiency is traded for the improved communications between the digital equipment and the human operator.

Decimal-to-BCD conversion is simple and straightforward. However, binary-to-BCD conversion is not direct. An intermediate conversion to decimal must be performed first. For example, the binary number 1011.01 is converted into its BCD equivalent.

First the binary number is converted to decimal.

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

Then the decimal result is converted to BCD.

$$11.25_{10} = 0001\ 0001.0010\ 0101$$

To convert from BCD to binary, the previous operation is reversed. For example, the BCD number 1001 0110.0110 0010 0101 is converted into its binary equivalent.

First, the BCD number is converted to decimal.

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10}$$

Then the decimal result is converted to binary.

$$96.625_{10} = 96_{10} + 0.625_{10}$$

$96 \div 2 = 48$	with remainder	0	← LSB
$48 \div 2 = 24$		0	
$24 \div 2 = 12$		0	
$12 \div 2 = 6$		0	
$6 \div 2 = 3$		0	
$3 \div 2 = 1$		1	
$1 \div 2 = 0$		1	← MSB

$$96_{10} = 1100000_2$$

$0.625 \times 2 = 1.25 = 0.25$	with overflow.	1	← MSB
$0.250 \times 2 = 0.50 = 0.50$		0	
$0.500 \times 2 = 1.00 = 0$		1	← LSB

$$0.625_{10} = 0.101_2$$

$$96.625_{10} = 96_{10} + 0.625_{10} = 1100000_2 + 0.101_2 = 1100000.101_2$$

Therefore:

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10} = 1100000.101_2$$

Because the intermediate decimal number contains both an integer and fraction, each number portion is converted as described under "Binary Number System." The binary sum (integer plus fraction) 1100000.101 is equivalent to the BCD number 1001 0110.0110 0010 0101.

DECIMAL	8421 BCD	GRAY	BINARY
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	0101
6	0110	0101	0110
7	0111	0100	0111
8	1000	1100	1000
9	1001	1101	1001
10	0001 0000	1111	1010
11	0001 0001	1110	1011
12	0001 0010	1010	1100
13	0001 0011	1011	1101
14	0001 0100	1001	1110
15	0001 0101	1000	1111

Figure 1-11

Codes.

0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

Special Binary Codes

Besides the standard pure binary coded form, the BCD numbering system is by far the most widely-used digital code. You will find one or the other in most of the applications that you encounter. However, there are several other codes that are used for special applications, such as the “Gray Code.”

The Gray Code is a widely-used, non-weighted code system. Also known as the cyclic, unit distance or reflective code, the Gray code can exist in either the pure binary or BCD formats. The Gray code is shown in Figure 1-11. As with the pure binary code, the first ten codes are used in BCD operations. Notice that there is a change in only one bit from one code number to the next in sequence. You can get a better idea about the Gray code sequence by comparing it to the standard 4-bit 8421 BCD code and the pure binary code also shown in Figure 1-11. For example, consider the change from 7 (0111) to 8 (1000) in the pure binary code. When this change takes place, all bits change. Bits that were 1's are changed to 0's and 0's are changed to 1's. Now notice the code change from 7 to 8 in the Gray code. Here 7 (0100) changes to 8 (1100). Only the first bit changes.

The Gray code is generally known as an error minimizing code because it greatly reduces confusion in the electronic circuitry when changing from one state to the next. When binary codes are implemented with electronic circuitry, it takes a finite period of time for bits to change from 0 to 1 or 1 to 0. These state changes can create timing and speed problems. This is particularly true in the standard 8421 codes where many bits change from one combination to the next. When the Gray code is used, however, the timing and speed errors are greatly minimized because only one bit changes at a time. This permits code circuitry to operate at higher speeds with fewer errors.

The biggest disadvantage of the Gray code is that it is difficult to use in arithmetic computations. Where numbers must be added, subtracted or used in other computations, the Gray code is not applicable. In order to perform arithmetic operations, the Gray code number must generally be converted into pure binary form.

Alphanumeric Codes

Several binary codes are called alphanumeric codes because they are used to represent characters as well as numbers. The two most common codes that will be discussed are ASCII and BAUDOT.

ASCII Code The American Standard Code for Information Interchange commonly referred to as ASCII, is a special form of binary code that is widely used in microprocessors and data communications equipment. A new name for this code that is becoming more popular is the American National Standard Code for Information Interchange (ANSII). However, this course will use the most recognized term, ASCII. ASCII is a 6-bit binary code that is used in transferring data between microprocessors and their peripheral devices, and in communicating data by radio and telephone. With six bits, a total of $2^6 = 64$ different characters can be represented. These characters comprise decimal numbers 0 through 9, upper-case letters of the alphabet, plus other special characters used for punctuation and data control. A 7-bit code called full ASCII, extended ASCII, or USASCII can be represented by $2^7 = 128$ different characters. In addition to the characters and numbers generated by 6-bit ASCII, 7-bit ASCII contains lower-case letters of the alphabet, and additional characters for punctuation and control. The 7-bit ASCII code is shown in Figure 1-12.

COLUMN		0 ⁽³⁾	1 ⁽³⁾	2 ⁽³⁾	3	4	5	6	7 ⁽³⁾
--------	--	------------------	------------------	------------------	---	---	---	---	------------------

ROW	BITS 4321 765	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	\	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(8	H	X	h	x
9	1001	HT	EM)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K		k	{
12	1100	FF	FS	,	<	L	\	l	!
13	1101	CR	GS	-	=	M		m	}
14	1110	SO	RS	.	>	N	~ ⁽¹⁾	n	~
15	1111	SI	US	/	?	O	— ⁽²⁾	o	DEL

Figure 1-12

Table of 7-bit American Standard Code
for Information Interchange.

NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) Explanation of special control functions in columns 0, 1, 2, and 7.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tabulation	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space (blank)	DEL	Delete

Figure 1-12
(Continued.)

The 7-bit ASCII code for each number, letter or control function is made up of a 4-bit group and a 3-bit group. Figure 1-13 shows the arrangement of these two groups and the numbering sequence. The 4-bit group is on the right and bit 1 is the LSB. Note how these groups are arranged in rows and columns in Figure 1-12.

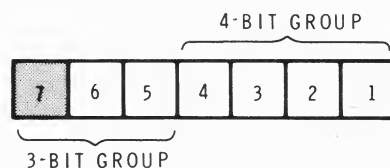


Figure 1-13

ASCII code word format.

To determine the ASCII code for a given number letter or control operation, locate that item in the table. Then use the 3- and 4-bit codes associated with the row and column in which the item is located. For example, the ASCII code for the letter L is 1001100. It is located in column 4, row 12. The most significant 3-bit group is 100, while the least significant 4-bit group is 1100. When 6-bit ASCII is used, the 3-bit group is reduced to a 2-bit group as shown in Figure 1-14.

In 7-bit ASCII code, an eighth bit is often used as a **parity** or check bit to determine if the data (character) has been transmitted correctly. The value of this bit is determined by the type of parity desired. **Even parity** means the sum of all the 1 bits, including the parity bit, is an even number. For example, if G is the character transmitted, the ASCII code is 1000111. Since four 1's are in the code, the parity bit is 0. The 8-bit code would be written 01000111.

OddParity means the sum of all the 1 bits, including the parity bit, is an odd number. If the ASCII code for G was transmitted with odd parity, the binary representation would be 11000111.

COLUMN		0	1	2	3	
ROW	BITS 4321	65	10	11	00	01
0	0000		SP ⁽³⁾	0	@	P
1	0001		!	1	A	Q
2	0010		"	2	B	R
3	0011		#	3	C	S
4	0100		\$	4	D	T
5	0101		%	5	E	U
6	0110		&	6	F	V
7	0111		'	7	G	W
8	1000		(8	H	X
9	1001)	9	I	Y
10	1010		*	:	J	Z
11	1011		+	;	K	
12	1100		,	<	L	\
13	1101		-	=	M]
14	1110		.	>	N	⌒ ⁽¹⁾
15	1111		/	?	O	— ⁽²⁾

Figure 1-14
Table of 6-bit American Standard Code
for Information Interchange.

NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) SP — Space (blank) for machine control.

BAUDOT Code While the ASCII code is used almost exclusively with microprocessor peripheral devices (CRT display, keyboard terminal, paper punch/reader, etc.), there are many older printer peripherals that use the 5-bit BAUDOT code. With five data bits, this code can represent only $2^5 = 32$ different characters. To obtain a greater character capability, 26 of the 5-bit codes are used to represent two separate characters. As shown in Figure 1-15, one set of 5-bit codes represents the 26 upper-case alphabet letters. The same 5-bit codes also represent various figures and the decimal number series 0 through 9.

The remaining six 5-bit codes are used for machine control and do not have a secondary function. Two of these 5-bit codes determine which of the 26 double (letter/figure) characters can be transmitted/received. Bit number 11111 forces the printer to recognize all following 5-bit codes as **letters**. Bit number 11011 forces **figure** recognition of all the following 5-bit codes. For example, to type 56 NORTH 10 STREET, the following method is used.

Type — Figures 5 6 Space

Then — Letters N O R T H Space

Then — Figures 1 0 Space

Finally — Letters S T R E E T

Bit Numbers 5 4 3 2 1	Letters Case	Figures Case
0 0 0 0 0	Blank	Blank
0 0 0 0 1	E	3
0 0 0 1 0	Line Feed	Line Feed
0 0 0 1 1	A	—
0 0 1 0 0	Space	Space
0 0 1 0 1	S	Bell
0 0 1 1 0	I	8
0 0 1 1 1	U	7
0 1 0 0 0	Car. Ret.	Car. Ret.
0 1 0 0 1	D	\$
0 1 0 1 0	R	4
0 1 0 1 1	J	(Apos)'
0 1 1 0 0	N	(Comma),
0 1 1 0 1	F	!
0 1 1 1 0	C	:
0 1 1 1 1	K	(
1 0 0 0 0	T	5
1 0 0 0 1	Z	"
1 0 0 1 0	L)
1 0 0 1 1	W	2
1 0 1 0 0	H	Stop
1 0 1 0 1	Y	6
1 0 1 1 0	P	0
1 0 1 1 1	Q	1
1 1 0 0 0	O	9
1 1 0 0 1	B	?
1 1 0 1 0	G	&
1 1 0 1 1	Figures	Figures
1 1 1 0 0	M	.
1 1 1 0 1	X	/
1 1 1 1 0	V	;
1 1 1 1 1	Letters	Letters

Figure 1-15
5-bit BAUDOT code table.

Self-Test Review

26. The BCD code is more convenient to use than the binary code because:
- A. it uses less bits.
 - B. it is more compatible with the decimal number system.
 - C. it is more adaptable to arithmetic computations.
 - D. there are more different coding schemes available.
27. Convert the following decimal numbers to 8421 BCD code.
- A. 1049
 - B. 267
 - C. 835
28. Convert the following 8421 BCD code numbers to decimal.
- A. 1001 0110 0010
 - B. 0111 0001 0100 0011
 - C. 1010 1001 1000
 - D. 1000 0000 0101
29. Convert the following binary numbers to 8421 BCD code.
- A. 101110.01
 - B. 1001.0101
 - C. 11011011.0001

30. Convert the following 8421 BCD codes to binary.
- A. 0001 1000 0010.0101
 - B. 0010 1001 0000.0010 0101
 - C. 1101 0110 0011.0101
 - D. 0110 1000.0001 0010 0101
31. Which code is best for error minimizing?
- A. 8421 BCD
 - B. pure binary
 - C. Gray
32. The ASCII and BAUDOT codes are a form of _____ codes.
33. To determine if the correct ASCII character has been transmitted, a _____ bit is often added to the code.
34. Which type of parity is used when the 8-bit ASCII character 01000111 is transmitted?
- A. odd
 - B. even
35. Refer to Figure 1-12 and convert the following characters into their ASCII 7-bit binary code.
- A. B
 - B. X
 - C. 3
 - D. S
36. Refer to Figure 1-12 and convert the following ASCII 7-bit binary codes to their character equivalents.
- A. 0110010
 - B. 1010110
 - C. 1011010
 - D. 1001110

Answers

26. B. More compatible with the decimal system.
27. A. 0001 0000 0100 1001
B. 0010 0110 0111
C. 1000 0011 0101
28. A. 962
B. 7143
C. Invalid (1010 represents a decimal digit greater than 9)
D. 805
29. A. $101110.01_2 = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1)$
 $+ (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$
 $= 32 + 0 + 8 + 4 + 2 + 0 + 0 + 0.25$
 $= 46.25_{10}$
 $46.25_{10} = 0100\ 0110.0010\ 0101$
 $101110.01_2 = 0100\ 0110.0010\ 0101$
B. $1001.0101_2 = 1001.0011\ 0001\ 0010\ 0101$
C. $11011011.0001_2 = 0010\ 0001\ 1001.0000\ 0110\ 0010\ 0101$

30. A. $0001\ 1000\ 0010.0101 = 182.5_{10}$
 $182.5_{10} = 182_{10} + 0.5_{10}$

$182 \div 2 = 91$	with remainder	0	← LSB
$91 \div 2 = 45$		1	
$45 \div 2 = 22$		1	
$22 \div 2 = 11$		0	
$11 \div 2 = 5$		1	
$5 \div 2 = 2$		1	
$2 \div 2 = 1$		0	
$1 \div 2 = 0$		1	← MSB

$$182_{10} = 10110110_2$$

$0.5 \times 2 = 1.00 = 0$	overflow	1	← MSB
			← LSB

$$0.5_{10} = 0.1_2$$

$$182.5_{10} = 182_{10} + 0.5_{10} = 10110110_2 + 0.1_2 = 10110110.1_2$$

$$0001\ 1000\ 0010.0101 = 10110110.1_2$$

- B. $0010\ 1001\ 0000.0010\ 0101 = 100100010.01_2$
 C. invalid (1101 represents a decimal digit greater than 9)
 D. $0110\ 1000.0001\ 0010\ 0101 = 1000100.001_2$

31. Gray

32. Alpha numeric

33. Parity

34. Even

35. A. 1000010
 B. 1011000
 C. 0110011
 D. 1010011

36. A. 2
 B. V
 C. Z
 D. N

EXPERIMENTS 1 AND 2

Perform Experiments 1 and 2 in the Experiment Section, Unit 7 of the course. After you finish the experiment, return to this unit and complete the "Unit Examination."

UNIT EXAMINATION

This examination will test your knowledge of the important facts in this unit. It will tell you what you have learned and what you need to review. Answer all questions first; then check your work against the correct answers given later.

1. Indicate the base or radix of the following number systems.

A. Octal 8.
 B. Decimal 10.
 C. Hexadecimal 16.
 D. Binary 2.

2. Write the following numbers using positional notation.

A. 1101.011_2 $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$
 B. 1010.01_{10} $1 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 0 \times 10^0 + 0 \times 10^{-1} + 1 \times 10^{-2}$
 C. 1001.101_8
 D. 1110.11_{16}

3. Convert the following numbers to decimal.

A. 10011.011_2 $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$
 B. 752.31_8 $7 \times 8^2 + 5 \times 8^1 + 2 \times 8^0 + 3 \times 8^{-1} + 1 \times 8^{-2}$
 C. $A8C.5F_{16}$ $10 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 + 5 \times 16^{-1} + 15 \times 16^{-2}$

4. Convert the following numbers to binary.

A. 105.0625_{10} 1101001.0001
 B. 374.24_8 111010110.001
 C. $F19.6C_{16}$ 11110001.10011011

5. Convert the following numbers to octal.

A. 638.3125_{10} 1174.49
 B. 10010101.0110101_2 225.321

6. Convert the following numbers to hexadecimal.

A. 9587.03125_{10} $2597.B$
 B. $1101101101010.101110101_2$ $156A.B0F$

7. The ASCII and BAUDOT codes are a form of _____ codes.
8. Convert the following numbers to 8421 BCD code.
- A. 521.372_{10}
- B. 1010.011_2
9. Convert the 8421 BCD code 1001 0101.0111 0011 to decimal.
10. Convert the 8421 BCD code 0101 0011.0111 0101 to binary.
11. Which type of parity is used when the 8-bit ASCII character 01110111 is transmitted?
- A. Odd
- B. Even
12. The BAUDOT code uses _____ bit numbers to generate a character.
13. Using only your knowledge of binary codes, identify the Gray code.

Decimal	A	B	C	D
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0011	0010	0010	0101
3	0010	0011	0011	0110
4	0110	0100	0100	0111
5	0111	0101	1011	1000
6	0101	0110	1100	1001
7	0100	0111	1101	1010
8	1100	1000	1110	1011
9	1101	1001	1111	1100

EXAMINATION ANSWERS

1. A. 8
B. 10
C. 16
D. 2
2. A. $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$
B. $(1 \times 10^3) + (0 \times 10^2) + (1 \times 10^1) + (0 \times 10^0) + (0 \times 10^{-1}) + (1 \times 10^{-2})$
C. $(1 \times 8^3) + (0 \times 8^2) + (0 \times 8^1) + (1 \times 8^0) + (1 \times 8^{-1}) + (0 \times 8^{-2}) + (1 \times 8^{-3})$
D. $(1 \times 16^3) + (1 \times 16^2) + (1 \times 16^1) + (0 \times 16^0) + (1 \times 16^{-1}) + (1 \times 16^{-2})$
3. A. 19.375_{10}
B. 490.390625_{10}
C. 2700.37109375_{10}
4. A. 1101001.0001_2
B. 11111100.0101_2
C. 111100011001.011011_2
5. A. 1176.24_8
B. 225.324_8
6. A. 2573.08_{16}
B. $1B6A.BA8_{16}$
7. Alpha numeric
8. A. 0101 0010 0001.0011 0111 0010
B. 0001 0000.0011 0111 0101
9. 95.73_{10}
10. 110101.11_2
11. B. Even
12. 5
13. A. Gray code

Unit 2

MICROCOMPUTER BASICS

CONTENTS

Introduction	2-3
Unit Objectives	2-4
Unit Activity Guide	2-5
Terms and Conventions	2-6
An Elementary Microcomputer	2-13
Executing a Program	2-27
Addressing Modes	2-43
Experiment 3	2-69
Unit Examination	2-71
Examination Answers	2-77

INTRODUCTION

A microprocessor is a very complex electronic circuit. It consists of thousands of microscopic transistors squeezed onto a tiny chip of silicon that is often no more than one-eighth inch square. The chip is placed in a package containing 40 or more leads.

The thousands of transistors that make up the microprocessor are arranged to form many different circuits within the chip. From the standpoint of learning how the microprocessor operates, the most important circuits on the chip are registers, counters, and decoders. In this unit, you will learn how these circuits can work together to perform simple but useful tasks.

UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Define the terms: microprocessor, microcomputer, input, output, I/O, I/O device, I/O port, instruction, program, stored program concept, word, byte, MPU, ALU, operand, memory, address, read, write, RAM, fetch, execute, MPU cycle, mnemonic, opcode, and bus.
2. Explain the purpose of the following circuits in a typical microprocessor: accumulator, program counter, instruction decoder, controller sequencer, data register, and address register.
3. Using a simplified block diagram of a hypothetical microprocessor, trace the data flow that takes place between the various circuits during the execution of a simple program.
4. Describe the difference between inherent, immediate, and direct addressing.
5. Write simple, straight-line programs that can be executed by the ET-6800 Microprocessor Trainer.

UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on Terms and Conventions.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 1 — 11.	_____
<input type="checkbox"/> Read the section on An Elementary Microcomputer.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 12 — 30.	_____
<input type="checkbox"/> Read the section on Executing a Program.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 31 — 40.	_____
<input type="checkbox"/> Read the section on Addressing Modes.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 41 — 50.	_____
<input type="checkbox"/> Perform Experiment 3.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

TERMS AND CONVENTIONS

A **microprocessor** is a logic device that is used in digital electronic systems. It is also being used by hobbyists, experimenters and low-budget research groups as a low-cost, general-purpose computer. But a distinction should be made between the microprocessor and the microcomputer.

The microprocessor unit, or MPU, is a complex logic element that performs arithmetic, logic, and control operations. The trend is to package it as a single integrated circuit.

A **microcomputer** contains a microprocessor, but it also contains other circuits such as memory devices to store information, interface adapters to connect it with the outside world, and a clock to act as a master timer for the system. Figure 2-1 shows a typical microcomputer in which these additional circuits are added. The arrows represent conductors over which binary information flows. The wide arrows represent several conductors connected in parallel. A group of parallel conductors which carry information is called a **bus**.

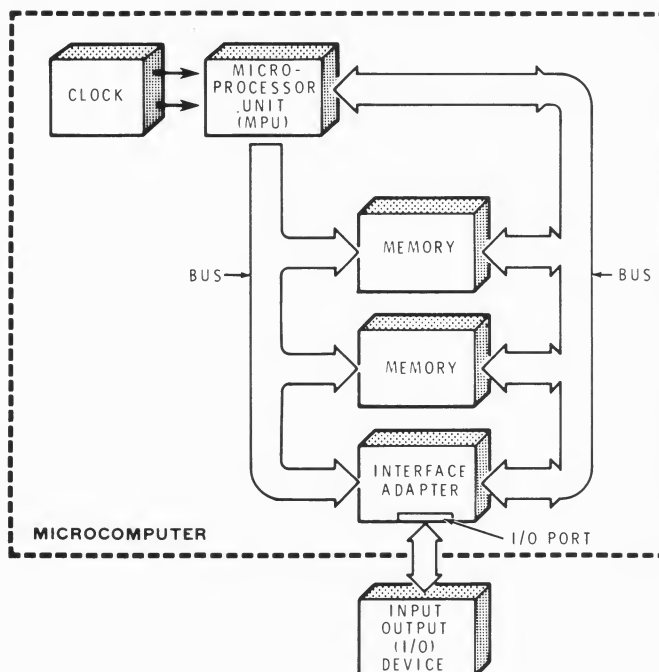


Figure 2-1
A Basic Microcomputer

The microcomputer is composed of everything inside the dotted line. Everything outside the dotted line is referred to as the **outside world**, and all microcomputers must have some means of communicating with it. Information received by the microcomputer from the outside world is referred to as **input** data. Information transmitted to the outside world from the microcomputer is referred to as **output** data.

Input information may come from devices like paper tape readers, typewriters, mechanical switches, keyboards, or even other computers. Output information may be sent to video displays, output typewriters, paper tape punches, or line printers. Some devices such as the teletypewriter can serve as both an input and an output device. These devices are referred to as **input/output** or **I/O** devices. The point at which the I/O device connects to the microcomputer is called an **I/O port**.

Stored Program Concept

A microcomputer is capable of performing many different operations. It can add and subtract numbers and it can perform logical operations. It can read information from an input device and transmit information to an output device. In fact, depending on the microprocessor used, there may be 100 or more different operations that the microcomputer can perform. Moreover, two or more individual operations can be combined to perform much more complex operations.

In spite of all its capabilities, the computer will do nothing on its own accord. It will do only what it is told to do, nothing more and nothing less. You must tell the computer exactly what operations to perform and the order in which it should perform them. The operations that the computer can be told to perform are called **instructions**. A few typical instructions are ADD, SUBTRACT, LOAD INDEX REGISTER, STORE ACCUMULATOR, and HALT.

A group of instructions that allow the computer to perform a specific job is called a **program**. One who writes these instructions is called a **programmer**. To design with microprocessors, the engineer must become a programmer. To repair microprocessor-based equipment, the technician must understand programming. Generally, the length of the program is proportional to the complexity of the task that the computer is to perform. A program for adding a list of numbers may require only a dozen instructions. On the other hand, a program for controlling all the traffic lights in a small city may require thousands of instructions.

A computer is often compared to a calculator, which is told what to do by the operator via the keyboard. Even inexpensive calculators can perform several operations that can be compared to instructions in the computer. By depressing the right keys, you can instruct the calculator to add, subtract, multiply, divide, and clear the display. Of course, you must also enter the numbers that are to be added, subtracted, etc. With the calculator, you can add a list of numbers as quickly as you can enter the numbers and the instructions. That is, the operation is limited by the speed and accuracy of the operator.

From the start, computer designers recognized that it was the human operator that slowed the computation process. To overcome this, the **stored program concept** was developed. Using this approach, the program is stored in the computer's memory. Suppose, for example, that you have 20 numbers that are to be manipulated by a program that is composed of 100 instructions. Let's further suppose that 10 answers will be produced in the process.

Before any computation begins, the 100-instruction program plus the 20 numbers are loaded into the computer's memory. Furthermore, 10 memory locations are reserved for the 10 answers. Only then is the computer allowed to execute the program. The actual computation time might be less than one millisecond. Compare this to the time that it would take to manually enter the instructions and numbers, one at a time, while the computer is running. This automatic operation is one of the features that distinguishes the computer from the calculator.

Computer Words

In computer terminology, a **word** is a group of binary digits that can occupy a storage location. Although the word may be made up of several binary digits, the computer handles each word as if it were a single unit. Thus, the **word** is the fundamental unit of information used in the computer.

A word may be a binary number that is to be handled as data. Or, the word may be an instruction that tells the computer which operation it is to perform. It may be an ASCII character representing a letter of the alphabet. Finally, a word can be an "address" that tells the computer where a piece of data is located.

Word Length

In the past few years, a wide variety of microprocessors have been developed. Their cost and capabilities vary widely. One of the most important characteristics of any microprocessor is the word length it can handle. This refers to the length in bits of the most fundamental unit of information.

The most common word length for microprocessors is 8 bits. In these units; numbers, addresses, instructions, and data are represented by 8-bit binary numbers. The lowest 8-bit binary number is 0000 0000₂ or 00₁₆. The highest is 1111 1111₂ or FF₁₆. In decimal, this range is from 0 to 255₁₀. Thus, an 8-bit binary number can have any one of 256₁₀ unique values.

An 8-bit word can specify positive numbers between 0 and 255₁₀. Or, if the 8-bit word is an instruction, it can specify any of 256₁₀ possible operations. It is also entirely possible that the 8-bit word is an ASCII character. In this case, it can represent letters of the alphabet, punctuation marks, or numerals. As you can see, the 8-bit word can represent many different things, depending on how it is interpreted. The programmer must insure that an ASCII character or binary number is not interpreted as an instruction. Later, you will see the consequences of making this mistake.

While the 8-bit word length is the most popular, other word lengths are sometimes used. The earliest microprocessor used a 4-bit word length, and four-bit microprocessors are still used in many cases because of their low cost. A few 12-bit and 16-bit microprocessors have also been developed.

Longer word lengths allow us to work with larger numbers. For example, a 16-bit word can represent numbers up to $65,535_{10}$. However, this capability adds to the complexity and cost of the microprocessor. Because most microprocessors use 8-bit word lengths, we will restrict our discussion to units of this type.

It should be pointed out that just because the word length is 8-bits, it does not mean that we are restricted to numbers below 256_{10} . It simply means that you must use two or more words to represent larger numbers.

The 8-bit word length defines the size of many different components in the microprocessor system. For example, many of the important registers will have 8-bit capacity. Memory will be capable of holding a large number of 8-bit words. And, the bus which is used to transfer data words will consist of eight parallel conductors.

Even 16-bit microprocessors use 8-bit segments of data in many applications. For example, inputs from teletypewriters often consist of 8-bit ASCII characters. To distinguish these 8-bit segments of information from the 16-bit (or longer) word lengths, another term has come into general use: the term **byte**. A byte is a group of bits that are handled as a single unit. Generally, a byte is understood to consist of 8-bits. In the 8-bit microprocessor, each word consists of one byte. But in the 16-bit machines, each word contains two bytes. Figure 2-2 illustrates these points.



Figure 2-2
Words and Bytes.

Figure 2-2 also shows how the bits that make up the computer word are numbered. The least significant bit (LSB) is on the right while the most significant bit (MSB) is on the left. In the 8-bit word, the bits are numbered 0 through 7 from right to left. In the 16-bit word, the bits are numbered 0 through 15 as shown. The lower 8-bits are called the lower order byte while the upper 8-bits are called the higher order byte.

Self-Test Review

1. Explain the difference between a microprocessor and a microcomputer. *Microprocessors may have more and devices that make constant with other devices*
2. What is a bus? *carries info in the computer*
3. Explain the difference between input and output data.
4. Define I/O.
5. The point at which data enters or leaves the computer is called an I/O port.
6. The operations that the computer can be told to perform are called instructions.
7. What is a program?
8. Explain what is meant by "stored program concept."
9. A byte generally consists of 8 bits.
10. A computer word may consist of one or more bytes.
11. What is the largest number that can be represented by an 8-bit computer word? By a 16-bit word? *255*
65535

Answers

1. A microprocessor is a logic element that can perform a variety of arithmetic, control, and logic operations. A microcomputer is a system that consists of a microprocessor, memory, interface adapters, clock, etc.
2. A bus is a group of conductors over which information can be transmitted. The conductors may be wires in a cable, foil patterns on a printed circuit board, or microscopic metal deposits in a silicon chip.
3. Input data is information that is entered into the computer from the outside world. Output data is information that is transmitted from the computer to the outside world. The outside world is defined as anything outside the computer.
4. I/O is the abbreviation for input-output. Thus, a device that can send data to and accept data from the computer is called an I/O device.
5. Port.
6. Instructions.
7. A program is a group of instructions that tell the computer the operations to be performed and the sequence in which they are to be performed.
8. The stored program concept refers to the technique of storing the instruction to be performed in the memory section along with the data that is to be operated upon.
9. 8.
10. Bytes.
11. $1111\ 1111_2$ or 255_{10} . $1111\ 1111\ 1111\ 1111_2$ or $65,535_{10}$.

AN ELEMENTARY MICROCOMPUTER

One of the difficulties you may encounter in learning about a microcomputer for the first time is the complexity of its main component — the microprocessor. The microprocessor may have a dozen or more registers varying in size from 1 bit to 16 bits. It will have scores of instructions, most of which can be implemented several different ways. It will have data, address, and control buses. In short, it can be very intimidating to start out by considering a full-blown microprocessor.

A better approach is to start with a “stripped down” version. By initially omitting some of the processor’s advanced features, we arrive at a device that can be readily understood and yet maintains the characteristics of an actual microprocessor. Strictly speaking, the microprocessor developed in this unit is hypothetical in nature. However, it is so close to the real thing that the programs we develop for it will actually run on the ET-6800 Microprocessor Trainer. Also, as you will see later, one of the most popular microprocessors in use today is a vastly advanced version of our elementary model.

A block diagram of a basic microcomputer is shown in Figure 2-3, which shows the microprocessor, the memory, and the I/O circuitry. For simplicity, we will ignore the I/O circuitry in this unit. We can do this by assuming that the program and data are already in memory and that the results of any computations will be held in a register or stored in memory. Ultimately of course, the program and data must come from the outside world and the results must be sent to the outside world. But we will save these procedures until a later unit. This will allow us to concentrate on the microprocessor unit and the memory.

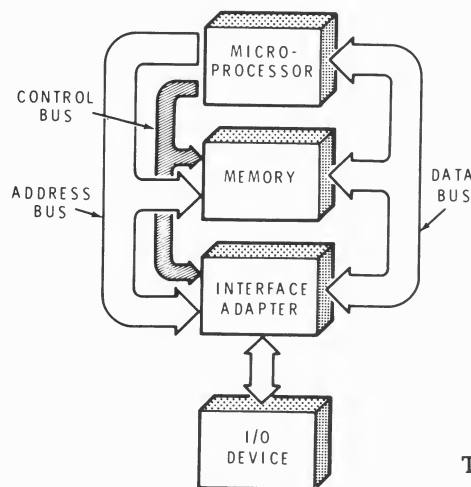


Figure 2-3
The Basic Microcomputer

The Microprocessor Unit (MPU)

The microprocessor unit is shown in greater detail in Figure 2-4. For simplicity, only the major registers and circuits are shown. In our elementary unit most of the counters, registers, and buses are 8-bits wide. That is, they can accommodate 8-bit words.

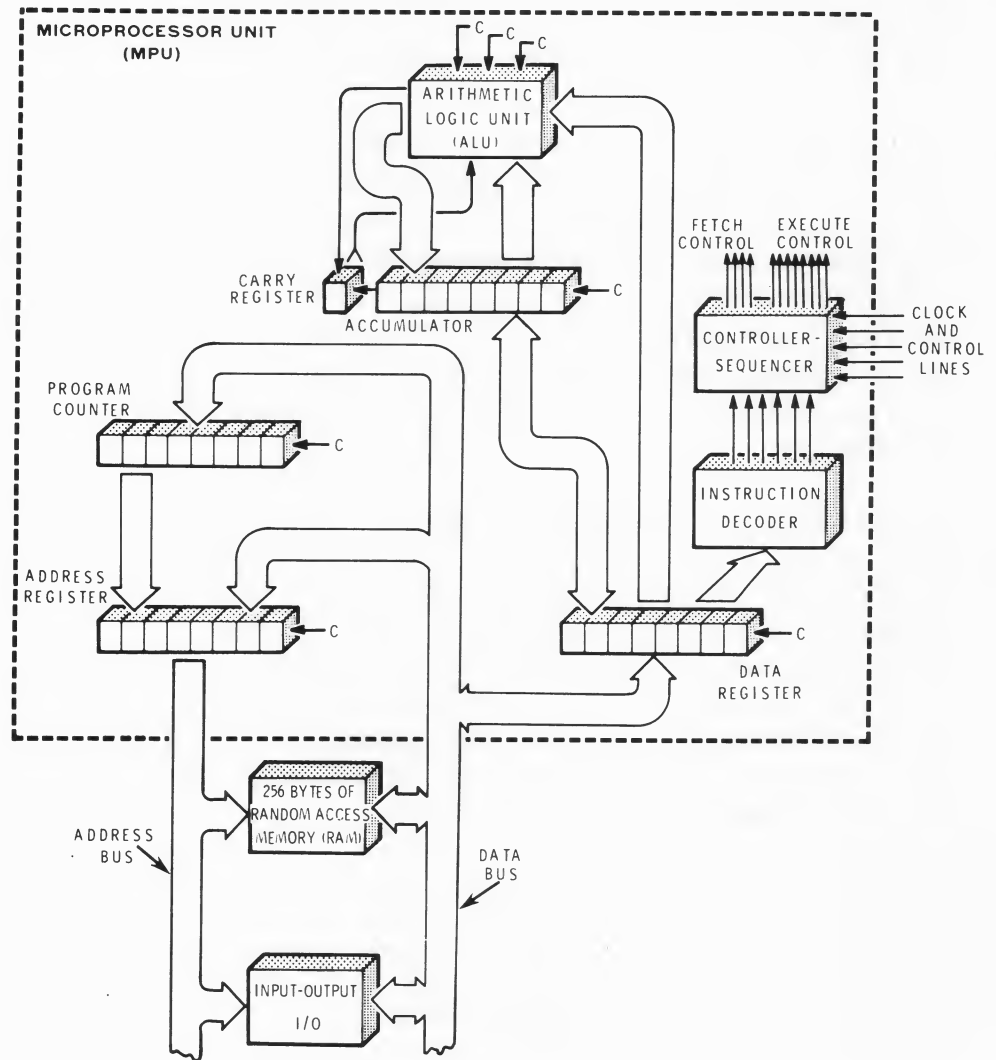


Figure 2-4

An Elementary Microprocessor.

One of the most important circuits in the microprocessor is the **arithmetic logic unit (ALU)**. Its purpose is to perform arithmetic or logic operations on the data words that are delivered to it. The ALU has two main inputs. One comes from a register called the accumulator, and the other comes from the data register. The ALU can add the two input data words together, or it can subtract one from the other. It can also perform some logic operations which will be discussed in later units. The operation that the ALU performs is determined by signals on the various control lines (marked C on the block diagram).

Generally, the ALU receives two 8-bit binary numbers from the accumulator and the data register as shown in Figure 2-5A. Because some operation is performed on these data words, the two inputs are called **operands**.

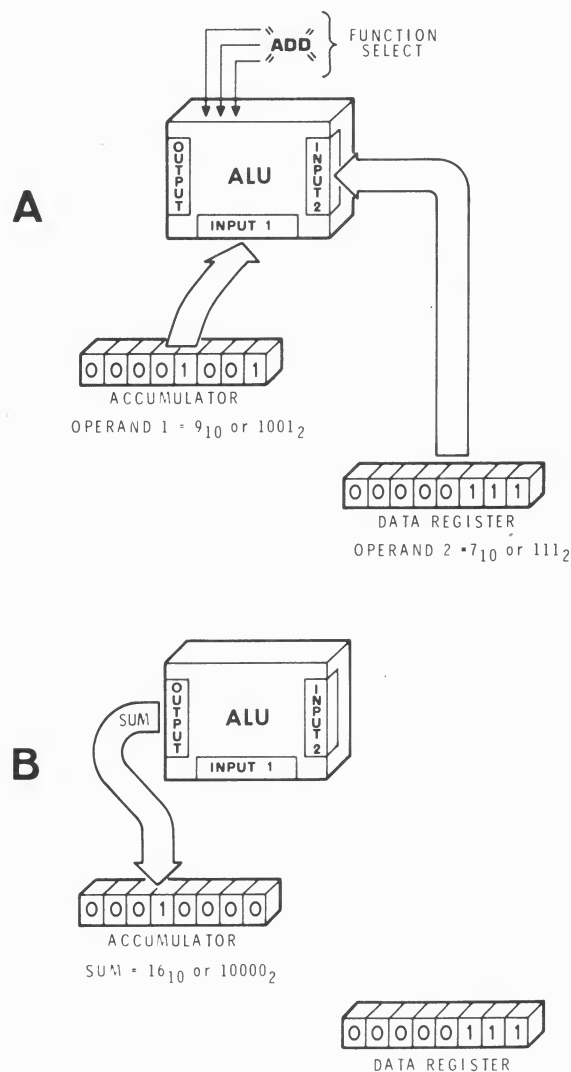


Figure 2-5
The Arithmetic Logic Unit.

The two operands may be added, subtracted, or compared in some way, and the result of the operation is stored back in the accumulator. For example, assume that two numbers (7 and 9) are to be added. Before the numbers can be added, one operand (9) is placed in the accumulator; the other (7) is placed in the data register. The proper control line is then activated to implement the add operation. The ALU adds the two numbers together, producing their sum (16_{10}) at the output. As shown in Figure 2-5B, the sum is stored in the accumulator, replacing the operand that was originally stored there. Notice that all the numbers involved are in binary form.

The **accumulator** is the most useful register in the microprocessor. During arithmetic and logic operations it performs a dual function. Before the operation, it holds one of the operands. After the operation, it holds the resulting sum, difference, or logical answer. The accumulator receives several instructions in every microprocessor. For example, the “load accumulator” instruction causes the contents of some specified memory location to be transferred to the accumulator. The “store accumulator” instruction causes the contents of the accumulator to be stored at some specified location in memory.

The **data register** is a temporary storage location for data going to or coming from the data bus. For example, it holds an instruction while the instruction is being decoded. Also, it holds a data byte while the word is being stored in memory.

The MPU also contains several other important registers and circuits: the address register, the program counter, the instruction decoder, and the controller-sequencer. These are shown in Figure 2-4.

The **address register** is another temporary storage location. It holds the address of the memory location or I/O device that is used in the operation presently being performed.

The **program counter** controls the sequence in which the instructions in a program are performed. Normally, it does this by counting in the sequence, 0, 1, 2, 3, 4, etc. At any given instant, the count indicates the location in memory from which the next byte of information is to be taken.

The **instruction decoder** does just what its name implies. After an instruction is pulled from memory and placed in the data register, the instruction is decoded by this circuit. The decoder examines the 8-bit code and decides which operation is to be performed.

The **controller-sequencer** produces a variety of control signals to carry out the instruction. Since each instruction is different, a different combination of control signals is produced for each instruction. This circuit determines the sequence of events necessary to complete the operation described by the instruction.

Later you will see how these various circuits work together to execute simple programs. But first, take a closer look at the memory for our microcomputer.

Memory

A simplified diagram of the 256-word, 8-bit read/write memory that is used in our hypothetical microcomputer is shown in Figure 2-6. The memory consists of 256_{10} locations, each of which can store an 8-bit word. This size memory is often referred to as 256×8 . A read/write memory is one in which data can be written in and read out with equal ease.

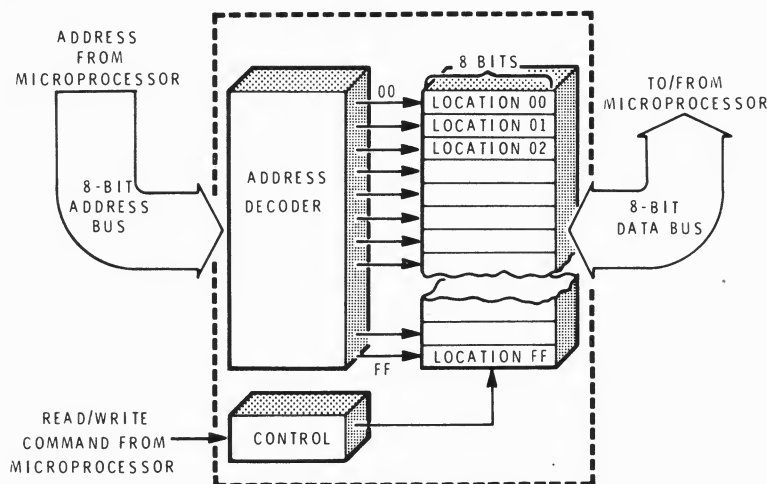


Figure 2-6
The Random Access Memory.

Two buses and a number of control lines connect the memory with the microprocessing unit. The address bus will carry an 8-bit binary number, which can specify 256_{10} locations, from the MPU to the memory address decoder. Each location is assigned a unique number called its address. The first location is given the address 0. The last location is given the address 255_{10} , which is 1111 1111 in binary and FF in hexadecimal. A specific location is selected by placing its 8-bit address on the address bus. The address decoder decodes the 8-bit number and selects the proper memory location.

The memory also receives a control signal from the MPU. This signal tells the memory the operation that is to be performed. A READ signal indicates that the selected location is to be read out. This means that the 8-bit number contained in the selected location is to be placed on the data bus where it can be transferred to the MPU.

The procedure is illustrated in Figure 2-7. Assume that the MPU is to read out the contents of memory location 04_{16} . Let's further assume that the number stored there is 97_{16} . First, the MPU places the address 04_{16} on the address bus. The decoder recognizes the address and selects the proper memory location. Second, the MPU sends a READ signal to the memory, indicating that the contents of the selected location are to be placed on the data bus. Third, the memory responds by placing the number 97_{16} on the data bus. The MPU can then pick up the number and use it as needed.

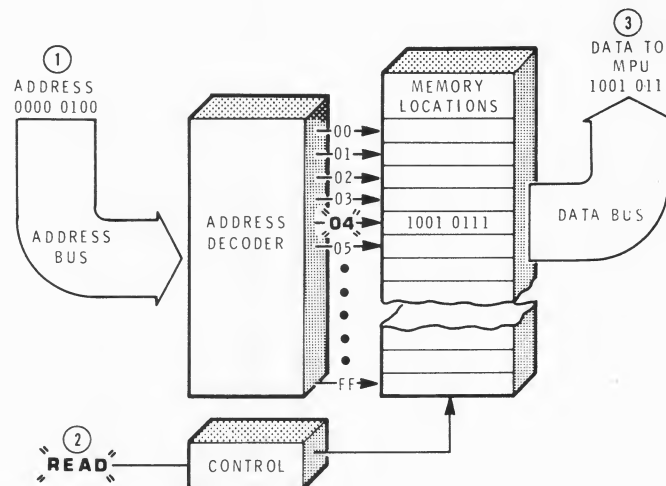


Figure 2-7
Reading from Memory.

It should be pointed out that the process of reading out a memory location does not disturb the contents of that location. That is, the number 97_{16} will still be present at memory location 04 after the read operation is finished. This characteristic is referred to as nondestructive readout (NDRO). It is an important feature because it allows us to read out the same data as many times as needed.

The MPU can also initiate a WRITE operation. This procedure is illustrated in Figure 2-8. During a WRITE operation, a data word is taken from the data bus and placed in the selected memory location. For example, let's see how the MPU can store the number 52_{16} at memory location 03. First, the MPU places the address 03 on the address bus. The decoder responds by selecting memory location 03. Second, the MPU places the number 52_{16} on the data bus. Third, the MPU sends the WRITE signal. The memory responds by storing the number contained on the data bus in the selected location. That is, 52_{16} is stored in location 03. The previous contents of the selected location are lost as the new number is written in that location.

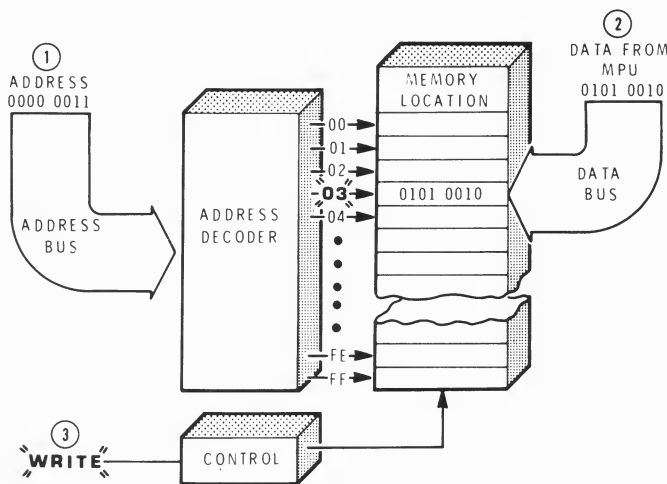


Figure 2-8
Writing into Memory.

The accepted name for a memory of this type is **Random Access Memory (RAM)**. "Random access" means that all memory locations are equally accessible. However, in recent years RAM has come to mean a random access **read/write** memory. As you will see later, there is another type of memory called a read only memory (ROM). It is also randomly accessible, but it does not have a write capability. Today, the accepted definition of RAM is a random access **read/write** memory. A read only memory, although it is randomly accessible, is called a ROM and never a RAM.

Fetch-Execute Sequence

When the microcomputer is executing a program, it goes through a fundamental sequence that is repeated over and over again. Recall that a program consists of instructions that tell the microcomputer exactly what operations to perform. These instructions must be stored in an orderly manner in memory. Instructions must be fetched, one at a time, from memory by the MPU. The instruction is then executed by the MPU.

The operation of the microcomputer can be broken down into two phases, as shown in Figure 2-9. When the microprocessor is initially started, it enters the **fetch phase**. During the fetch phase, an instruction is taken from memory and decoded by the MPU. Once the instruction is decoded, the MPU switches to the **execute phase**. During this phase, the MPU carries out the operation dictated by the instruction.

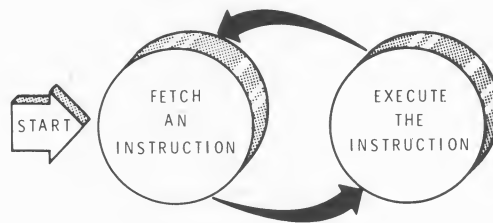


Figure 2-9
The Fetch-Execute Sequence.

The fetch phase always consists of the same series of operations. Thus, it always takes the same amount of time. However, the execute phase will consist of different sequences of events, depending on what type of instruction is being executed. Thus, the time of the execute phase may vary considerably from one instruction to the next.

A Sample Program

Now that you have a general idea of the registers and circuits found in a microcomputer, we will examine how all these circuits work together to execute a simple program. At this point, we are primarily interested in showing you how the microcomputer operates. Therefore, the program will be a very trivial one.

Let's see how the computer goes about solving a problem like $7 + 10 = ?$ While this seems like an incredibly easy problem, the computer, if left to its own resources, does not have the foggiest notion of how to solve it. You must tell the computer how to solve the problem right down to the smallest detail. You do this by writing a program.

Before you can write the program you must know what instructions are available to you and the computer. Every microprocessor comes with a listing of its instruction set. Assume that, after looking over the list, you decide that three instructions are necessary to solve the problem. These three instructions and a description of what they do are shown in Figure 2-10.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	1000 0110 ₂ or 86 ₁₆	Load the contents of the next memory location into the accumulator.
Add	ADD	1000 1011 ₂ or 8B ₁₆	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Halt	HLT	0011 1110 ₂ or 3E ₁₆	Stop all operations.

Figure 2-10

Instructions Used in the Sample Program.

The first column in the table gives the name of the instruction. When writing programs, it is often inconvenient to write out the entire name. For this reason, each instruction is given an abbreviation or a memory aid called a **mnemonic**. The mnemonics are given in the second column. The third column is called the operation code or **opcode**. This is the binary number that the computer and the programmer use to represent the instruction. The opcode is given in both binary and hexadecimal form. The final column describes exactly what operation is performed when the instruction is executed. Study this table carefully; you will be using these instructions over and over again.

Assume that you wish to add 7 to 10_{10} and place the sum in the accumulator. The program is an elementary one. First you will load 7 into the accumulator with the LDA instruction. Next, you will add 10_{10} to the accumulator using the ADD instruction. Finally, you will stop the computer with the HLT instruction.

Using the mnemonics and the decimal representation of the numbers to be added, the program looks like this:

```
LDA 7
ADD 10
HLT
```

Unfortunately, the basic microcomputer cannot understand mnemonics or decimal numbers. It can interpret binary numbers and nothing else. Thus, you must write the program as a sequence of binary numbers. You can do this by replacing each mnemonic with its corresponding opcode and each decimal number with its binary counterpart.

That is:

LDA 7	becomes	1000 0110	0000 0111
		opcode from	binary representation
		Figure 2-10	for 7

And:

ADD 10	becomes	1000 1011	0000 1010
		opcode from	binary representation
		Figure 2-10	for 10_{10}

Finally,

HLT	becomes	0011 1110
		opcode from
		Figure 2-10

Notice that the program consists of three instructions. The first two instructions have two parts: an 8-bit opcode followed by an 8-bit operand. The operands are the two numbers that are to be added (7 and 10_{10}).

Recall that the microprocessor and memory work with 8-bit words or bytes. Because the first two instructions consist of 16-bits of information, they must be broken into two 8-bit bytes before they can be stored in memory. Thus, when the program is stored in memory, it will look like this:

1st Instruction	{ 1000 0110	Opcode for LDA
	{ 0000 0111	Operand (7)
2nd Instruction	{ 1000 1011	Opcode for ADD
	{ 0000 1010	Operand (10 ₁₀)
3rd Instruction	0011 1110	Opcode for HLT

Five bytes of memory are required. You can store this 5-byte program any place in memory you like. Assuming you store it at the first five memory locations, the memory can be diagrammed as shown in Figure 2-11.

ADDRESS		MEMORY	MNEMONICS/CONTENTS
HEX	BINARY	BINARY CONTENTS	
00	0000 0000	1 0 0 0 0 1 1 0	LDA
01	0000 0001	0 0 0 0 0 1 1 1	7
02	0000 0010	1 0 0 0 1 0 1 1	ADD
03	0000 0011	0 0 0 0 1 0 1 0	10 ₁₀
04	0000 0100	0 0 1 1 1 1 1 0	HLT
	.		
	.		
	.		
	.		
	.		
FD	1111 1101		
FE	1111 1110		
FF	1111 1111		

Figure 2-11

The Program in Memory.

Notice that each memory location has two 8-bit binary numbers associated with it. One is its address, the other is its contents. Be careful not to confuse these two numbers. The address is fixed. It is established when the microcomputer is built. However, the contents may be changed at any time by storing new data.

Before you see how this program is executed, let's review the material covered in this section.

Self-Test Review

12. The circuit in the microprocessor that performs arithmetic and logic operations is called the ALU.
13. The numbers that are operated upon by the computer are called operands.
14. Where are the two operands held as they are transferred to the ALU?
ACCUMULATOR
DATA Register
15. Where is the result held after an arithmetic operation?
ACCUMULATOR
16. Which register in the MPU holds the instruction while the opcode is being decoded?
DATA Register
17. Which circuit in the MPU determines the memory location from which the next byte of information will be taken?
PROGRAM COUNTER
18. Which register holds the address while it is being decoded?
ADDRESS REGISTER
19. How many memory locations can be specified by an 8-bit address?
256
20. Explain the 3-step procedure for writing the number 34_{16} into memory location 9.
ADDRESS
DATA
WRITE
21. Define mnemonic, opcode, and RAM.
22. An instruction is retrieved from memory and decoded during the fetch phase, and the operation indicated by this instruction is carried out during the DECODE phase.

23. In our hypothetical microcomputer, the mnemonic for the load accumulator instruction is LDA.
24. The opcode for the halt instruction is 0011 110.
25. When the ADD instruction is executed, where is the SUM stored?
ACCUMULATOR
26. How many memory locations are required to store the following program?
4
- LDA 13_{10}
ADD 17_{10}
ADD 10_{10}
HLT
27. What will be the contents of the accumulator after this program is executed?
40

Answers

12. Arithmetic logic unit (ALU).
13. Operands.
14. One is held in the accumulator; the other in the data register.
15. In the accumulator.
16. Data register.
17. The program counter.
18. The address register.
19. $2^8 = 256_{10}$.
20. Step 1. The address 9 is placed on the address bus.
Step 2. The data 34_{16} is placed on the data bus.
Step 3. The read/write line is switched to the write state.
21. A mnemonic is an abbreviation or memory aid.

An opcode is a binary number that tells the microprocessor which instruction to execute.

RAM has come to mean a random access read/write memory.
22. Fetch, execute.
23. LDA.
24. $0011\ 1110_2$.
25. In the accumulator.
26. Seven.
27. 40_{10} or 101000_2 .

EXECUTING A PROGRAM

Before a program can be run, it must be placed in memory. Later, you will see how this is done. For now, assume that we have already loaded the program developed in the previous section.

The important registers of the microcomputer are shown in Figure 2-12. Notice that our 5-byte program for adding 7 and 10₁₀ is shown in memory. The following paragraphs will take you through the step-by-step procedure by which the computer executes this program.

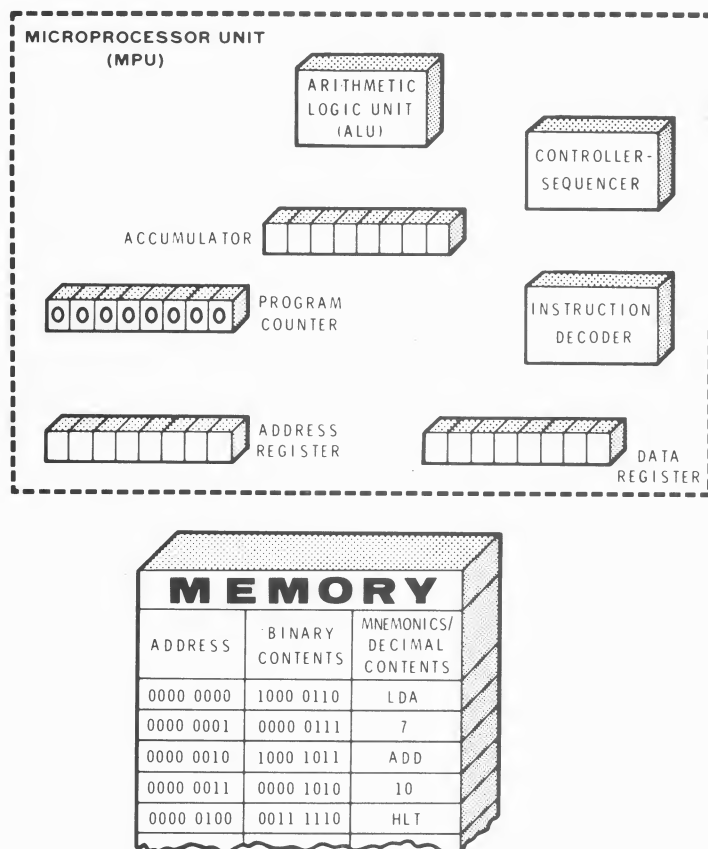


Figure 2-12

The Program Counter is Set to the Address of the First Instruction.

To begin executing the program, the program counter must be set to the address of the first instruction. In this case, the first instruction is in memory location 0000 0000, so the program counter is set accordingly. The procedure for setting the program counter to the proper address will be discussed later.

The Fetch Phase

The first step is to fetch the first instruction from memory. The sequence of events that happen during the fetch phase is controlled by the controller-sequencer. It produces a number of control signals which will cause the events illustrated in Figure 2-13 through 2-17 to occur.

First the contents of the program counter are transferred to the address register as shown in Figure 2-13. Recall that this is the address of the first instruction.

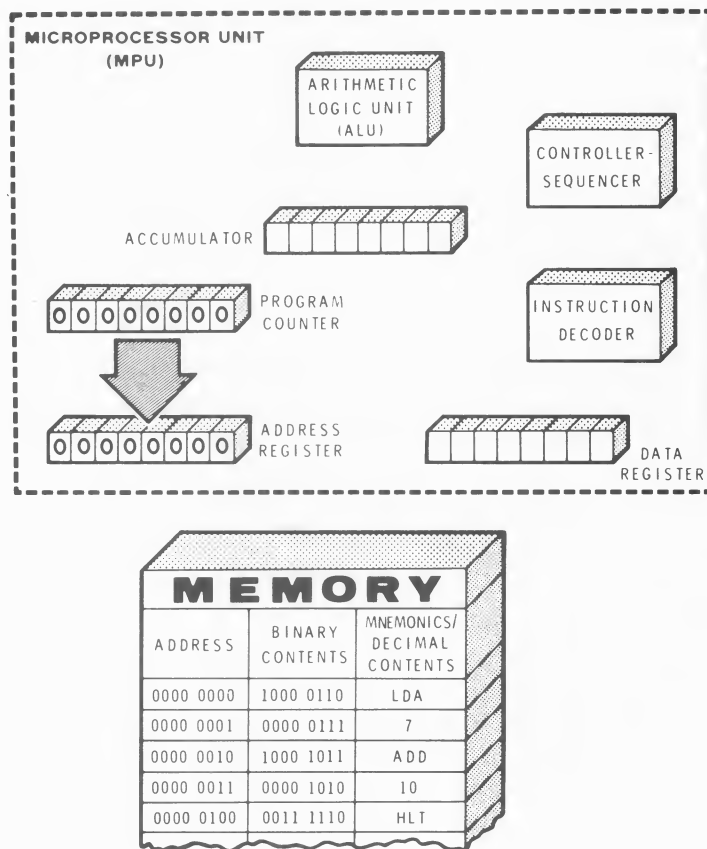


Figure 2-13

The Contents of the Program Counter are Transferred to the Address Register.

Once the address is safely in the address register, the program counter is incremented by one. That is, its contents change from 0000 0000 to 0000 0001 (Figure 2-14). Notice that this does not change the contents of the address register in any way.

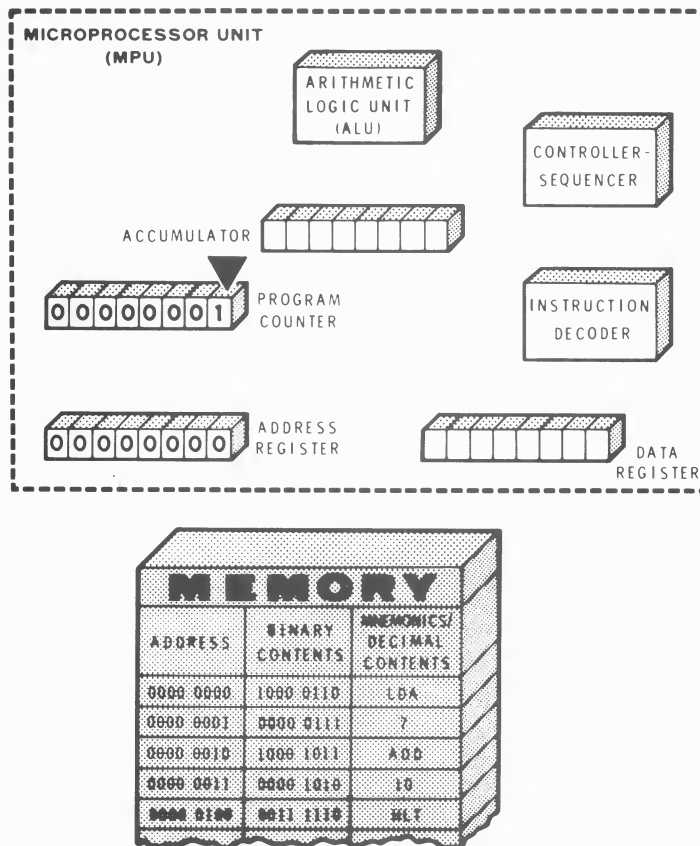


Figure 2-14
The Program Counter is Incremented.

Next, the contents of the address register (0000 0000) are placed on the address bus (Figure 2-15). The memory circuits decode the address and select memory location 0000 0000.

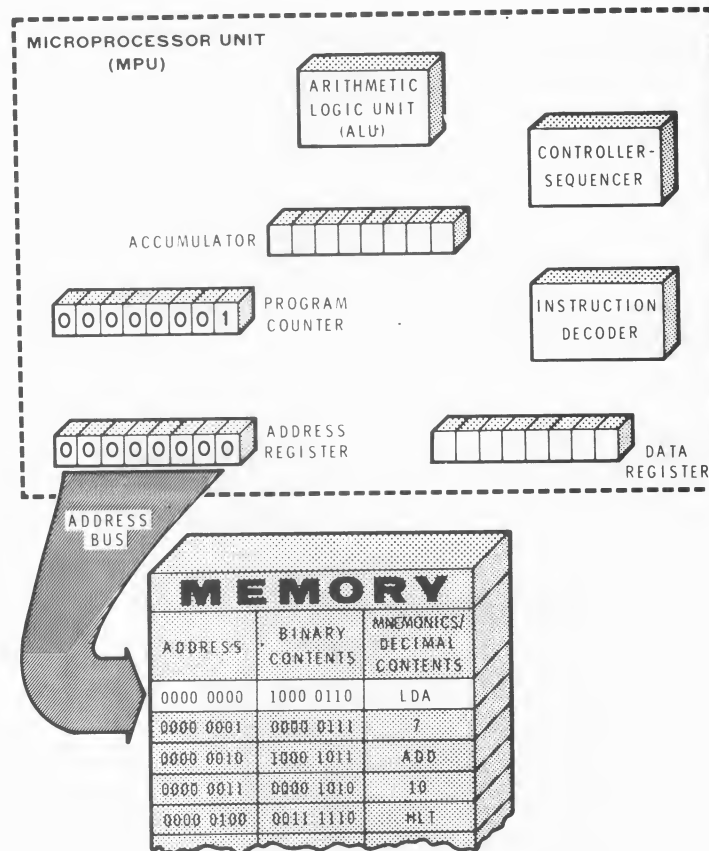


Figure 2-15
The Address of the First Instruction is Placed on the Address Bus.

The contents of the selected memory location are placed on the data bus and transferred to the data register in the MPU. After this operation, the opcode for the LDA instruction will be in the data register as shown in Figure 2-16.

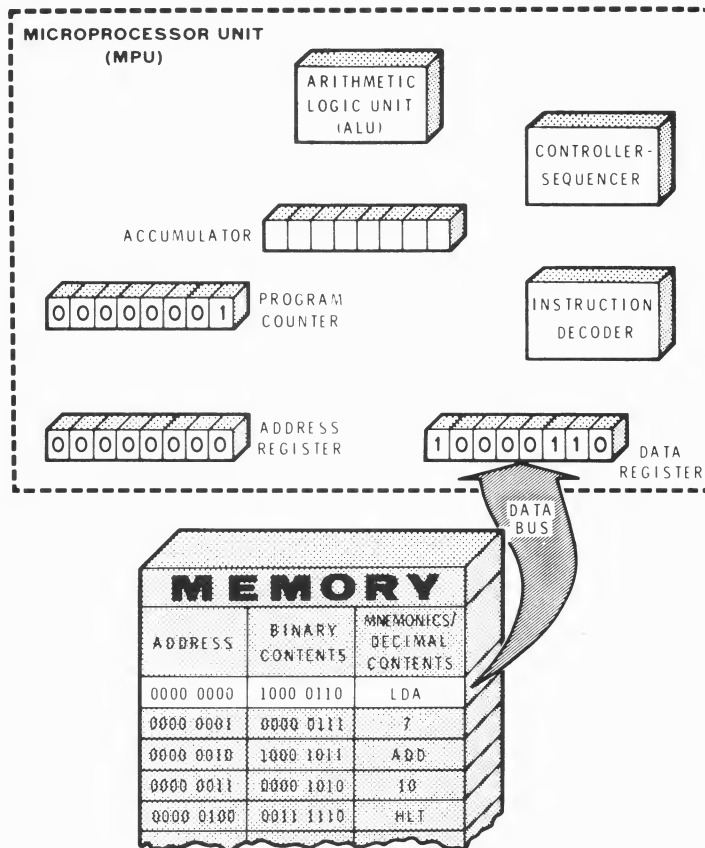


Figure 2-16
The Opcode of the First Instruction is
Placed on the Data Bus.

The next step is to decode the instruction (Figure 2-17). The opcode is transferred to the instruction decoder. This circuit recognizes that the opcode is that of an LDA instruction. It informs the controller-sequencer of this fact and the sequencer produces the necessary control pulses to carry out the instruction. This completes the fetch phase of the first instruction.

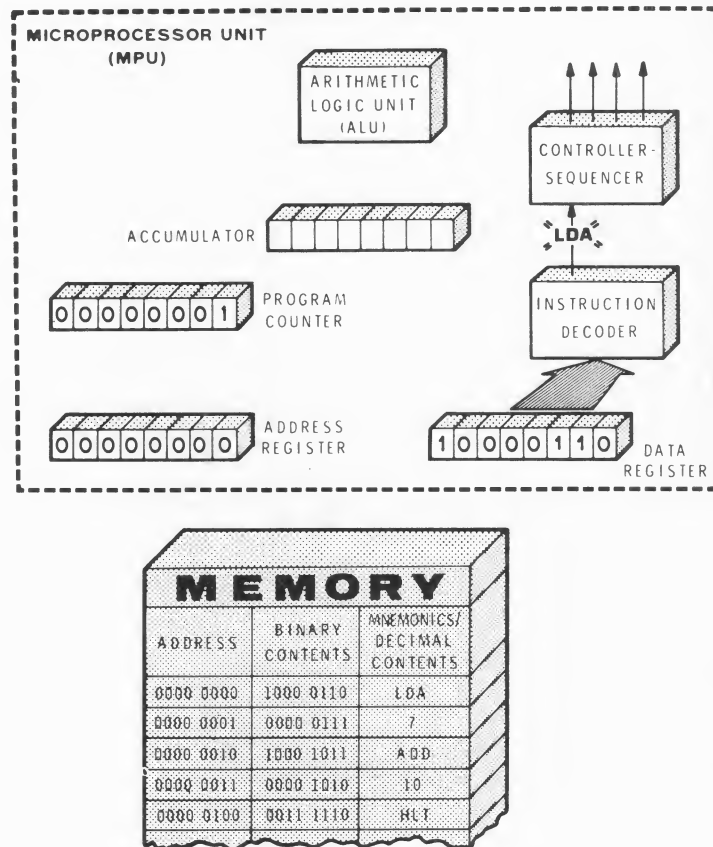


Figure 2-17
The Opcode is Decoded.

The Execute Phase

The first instruction was fetched from memory and decoded during the fetch phase. The MPU now knows that this is an LDA instruction. During the execute phase, it must carry out this instruction by reading out the next byte of memory and placing it in the accumulator.

The first step is to transfer the address of the next byte from the program counter to the address register (Figure 2-18). You will recall that the program counter was incremented to the proper address (0000 0001) during the previous fetch phase.

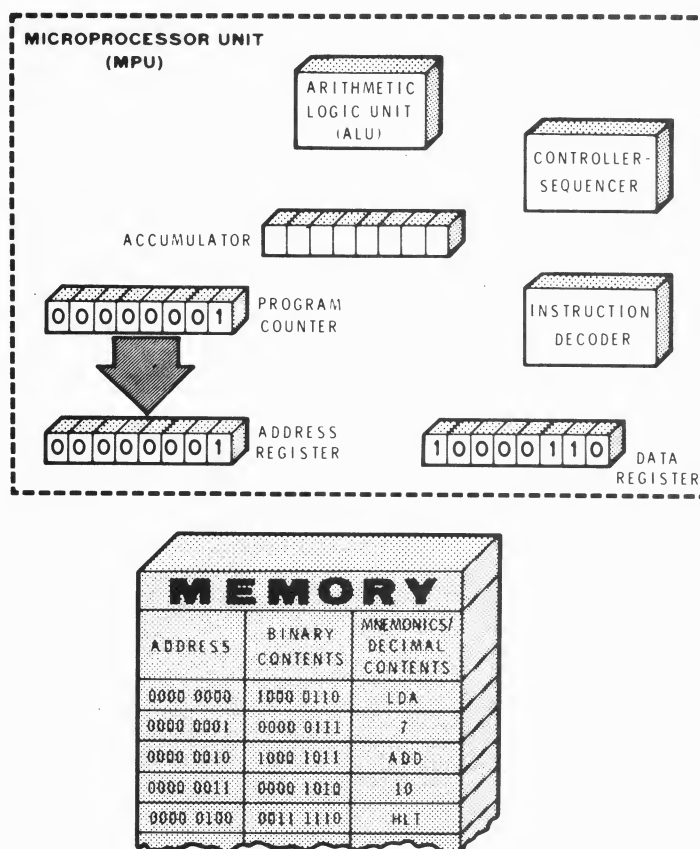


Figure 2-18
The Contents of the Program Counter
are Transferred to the Address Register.

The next two operations are shown in Figure 2-19. First, the program counter is incremented to 0000 0010 in anticipation of the next fetch phase. Second, the contents of the address register (000 0001) are placed on the address bus.

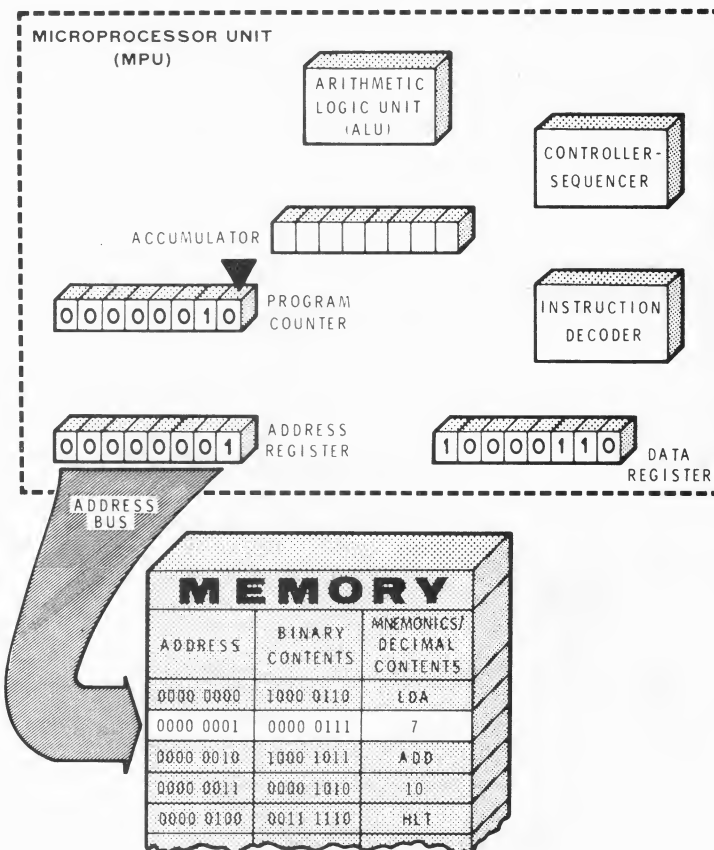


Figure 2-19

The Program Counter is Incremented;
the Contents of the Address Register
are Placed on the Address Bus.

The address is decoded and the contents of memory location 0000 0001 are loaded into the data register as shown in Figure 2-20. Recall that this is the number 7. An instant later, the number is transferred to the accumulator. Thus, the first execute phase ends with the number 7 in the accumulator.

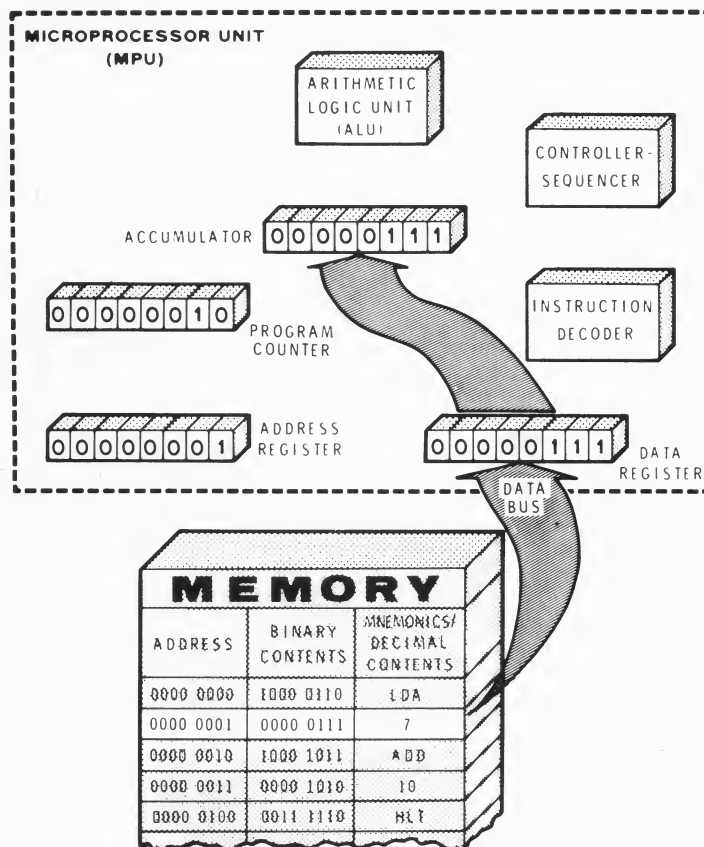


Figure 2-20
The First Operand is Transferred to the Accumulator Via the Data Register.

Fetching the Add Instruction

The next instruction in our program is the ADD instruction. It is fetched from memory using the same procedure outlined above. Figure 2-21 illustrates this five-step procedure:

1. The contents of the program counter (0000 0010) are transferred to the address register.

2. The program counter is incremented to 0000 0011.
3. The address is placed on the address bus.
4. The contents of the selected memory location are transferred to the data register.
5. The contents of the data register are decoded by the instruction decoder.

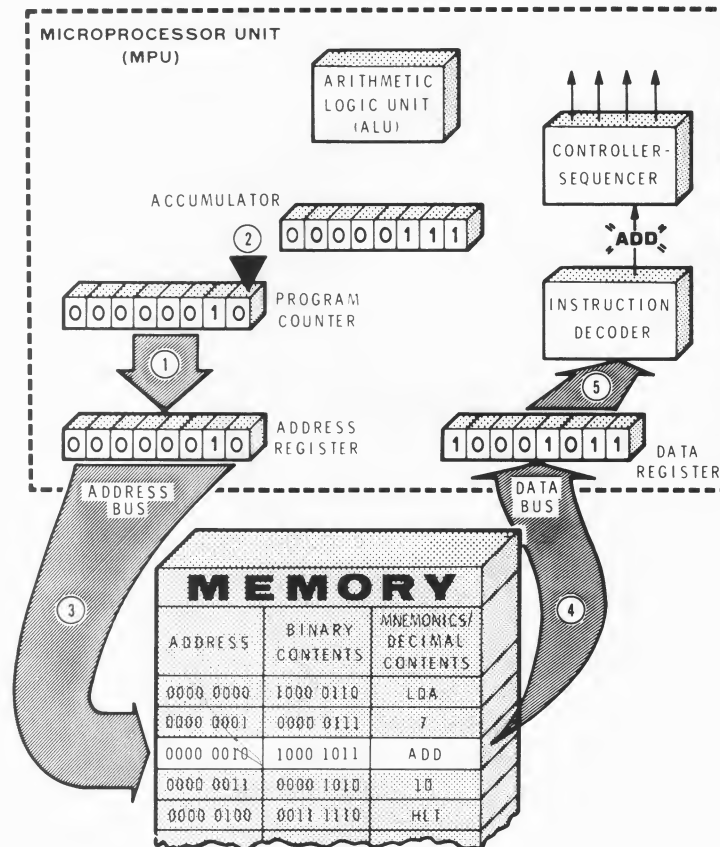


Figure 2-21

Fetching the ADD Instruction.

The data word fetched from memory is the opcode for the ADD instruction. Thus, the controller-sequencer produces the necessary control pulses to execute this instruction.

Executing the Add Instruction

The execution of the ADD instruction is a six-step procedure. This procedure is illustrated in Figure 2-22.

1. The contents of the program counter (0000 0011) are transferred to the address register.
2. The program counter is incremented to 0000 0100 in anticipation of the next fetch phase.
3. The address of the operand is placed on the address bus.
4. The operand (10_{10}) is transferred to the data register.
- 5A. The operand (10_{10}) is transferred into one input of the ALU.
- 5B. Simultaneously, the other operand (7) is transferred from the accumulator to the other input of the ALU.
6. The ALU adds the two operands. Their sum ($0001\ 0001_2$ or 17_{10}) is loaded into the accumulator, destroying the number (7) that was previously stored there.

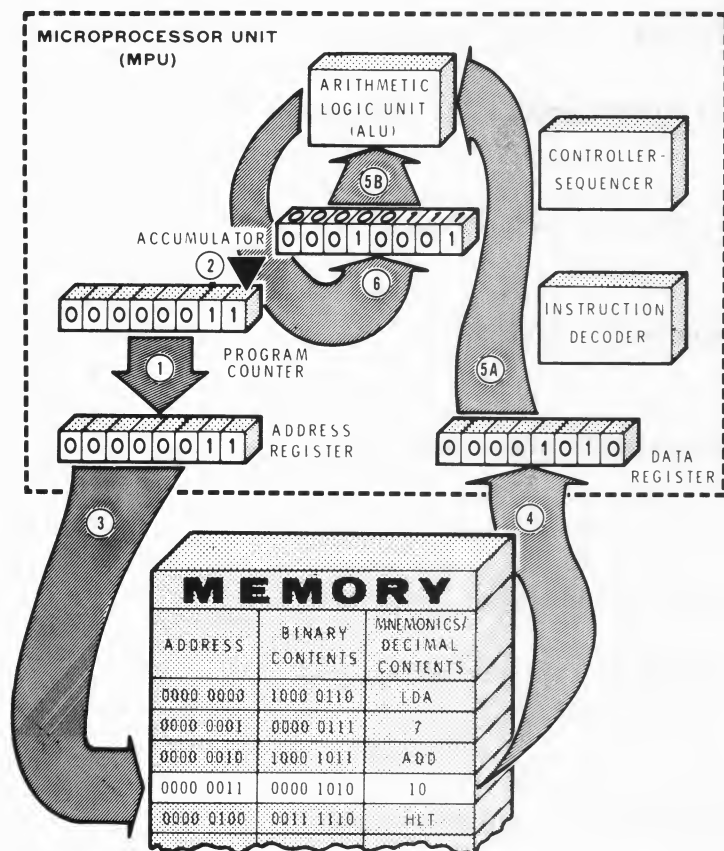


Figure 2-22
Executing the ADD Instruction.

The computation portion of our program ends with the sum of the two operands in the accumulator. However, the program is not finished until it tells the computer to stop executing instructions.

Fetching and Executing the HLT Instruction

The final instruction in the program is a HLT instruction. It is fetched using the same fetch procedure as before. The five steps are illustrated in Figure 2-23. The address comes from the program counter via the address register. When this address is placed on the address bus, memory location 0000 0100 is read out and the opcode for HLT is loaded into the data register. The opcode is decoded and the instruction is executed.

The execution of the HLT instruction is very simple. The controller-sequencer simply stops producing control signals. Consequently, all computer operations stop. Notice that the program has accomplished our objective of adding 7_{10} to 10_{10} . The resulting sum, 17_{10} , is in the accumulator.

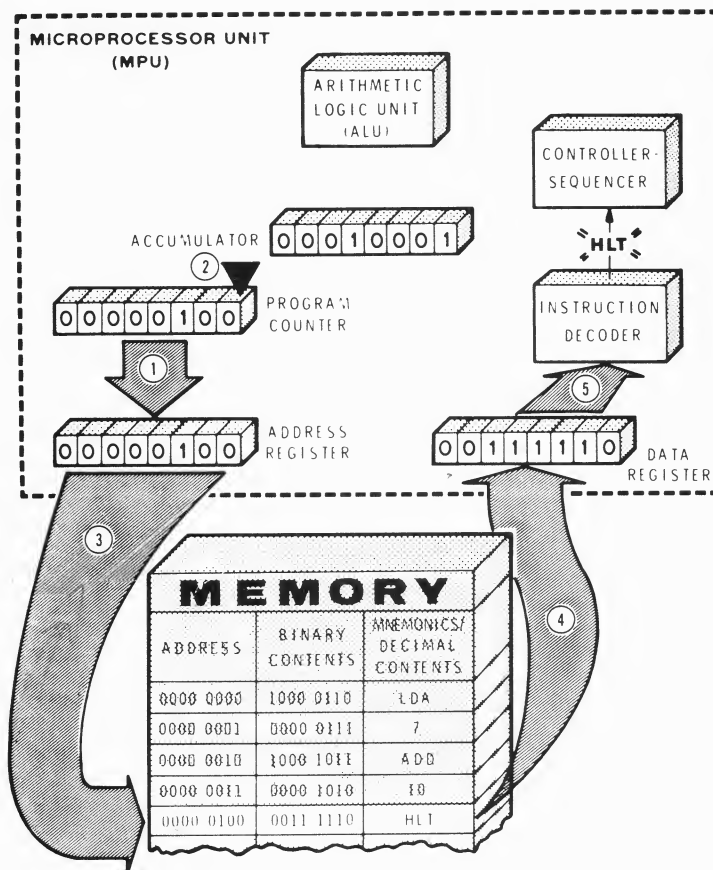


Figure 2-23
Fetching and Executing the HLT Instruction.

Self-Test Review

Examine this sample program carefully and answer the questions below:

```
LDA 8  
ADD 7  
ADD 4  
HLT
```

31. During the first fetch phase, what binary number is loaded into the data register? *1000 0110*
32. During the first execute phase the number $0000\ 1000_2$ is loaded into the _____.
33. During the second fetch phase, what binary number is loaded into the data register? *0000*

34. If the first byte of the program is placed in address 0000 0000, what is the address of the first ADD instruction? 0000 0011
35. How many bytes of memory are taken up by the program? 7
36. What number is in the accumulator during the third fetch phase? 1000 0001
37. When the program is finished running, what number is in the accumulator? 17
38. What is the final number in the program counter?
39. What are the final contents of the address register?
40. What are the final contents of the data register?

Answers

- 31. The opcode for the LDA instruction, $1000\ 0110_2$.
- 32. Accumulator.
- 33. The opcode for ADD, $1000\ 1011$.
- 34. $0000\ 0010_2$.
- 35. Seven.
- 36. 15_{10} or $0000\ 1111_2$.
- 37. 19_{10} or $0001\ 0011_2$.
- 38. 7_{10} or $0000\ 0111_2$.
- 39. 6_{10} or $0000\ 0110_2$.
- 40. The opcode for the HLT instruction, $0011\ 1110_2$.

ADDRESSING MODES

If you examine the program discussed in the previous section, you will find that it uses two distinctly different types of instructions. One type of instruction requires an operand. LDA and ADD are examples of this type. These are two-byte instructions. The first byte is the opcode; the second is the operand.

Microprocessors also have single-byte instructions. HLT is a good example. This instruction requires no operand; thus, it can be implemented with a single byte.

Instructions can be classified in several different ways. One of the most basic distinctions is their addressing mode. The addressing mode refers to the method by which an instruction addresses its operand.

Inherent or Implied Addressing

Single-byte instructions have no operand or the operand is implied by the opcode itself. For example, the HLT instruction has no operand at all. However, some single-byte instructions may have an implied operand. An example is the “increment accumulator” instruction. The number that is operated upon (incremented) is the number in the accumulator. This instruction simply adds one to the contents of the accumulator. This type of addressing will be referred to in this course as **implied addressing** or **inherent addressing**. The operations performed during an instruction of this type are illustrated in Figure 2-24.

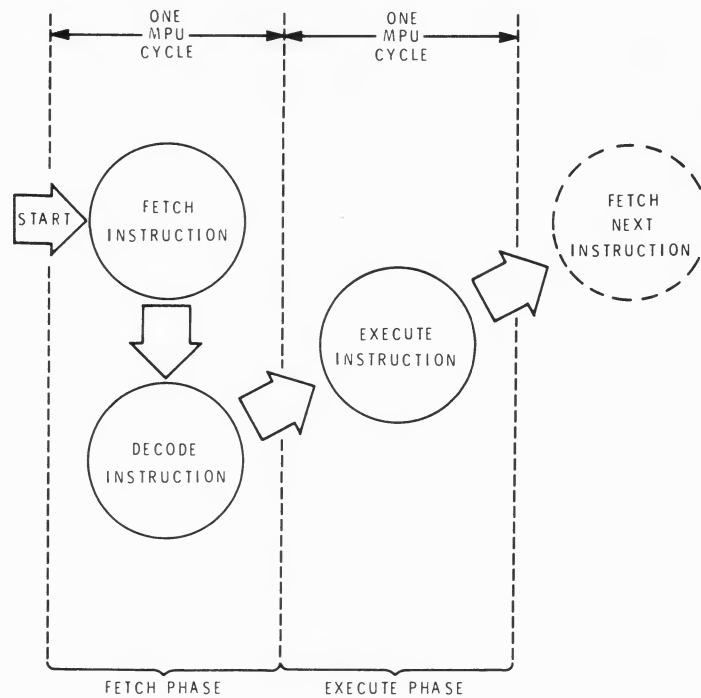


Figure 2-24
Operations Performed in the Inherent
or Implied Addressing Mode.

Immediate Addressing

In our previous program, the two-byte instructions use the **immediate addressing** mode. In this mode, the operand is the byte immediately following the opcode. That is, the byte of data that is to be operated upon is the second byte of the instruction. When these two-byte instructions are stored in memory, the address of the operand is the memory location following the opcode. The operations performed during this type of instruction are illustrated in Figure 2-25.

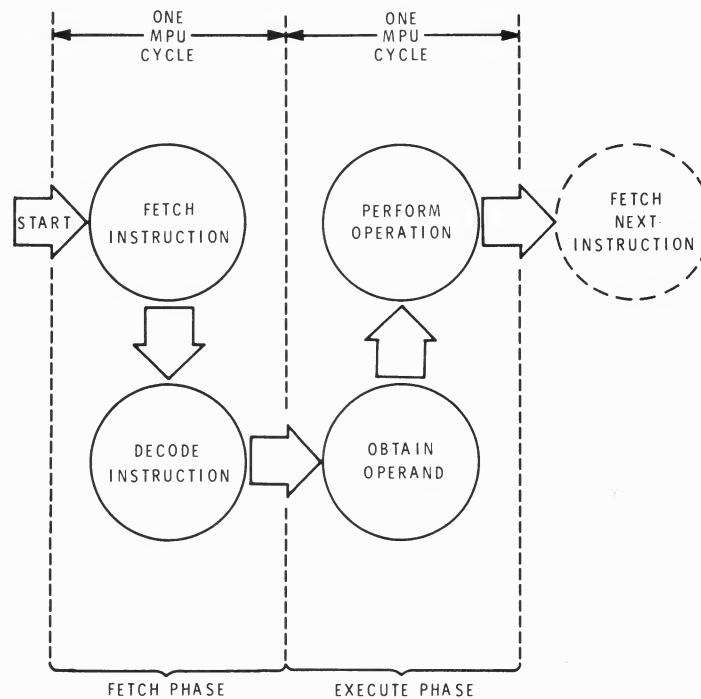


Figure 2-25
Operations Performed in the Im-
mediate Addressing Mode.

The inherent and immediate addressing modes have two advantages. First, they require little memory space; one and two bytes respectively. This is important because memory locations cost money. Generally, the less memory space taken by a program, the better off we are. Second, these addressing modes require a minimum of execution time. The execution time can become important in long programs.

The time required for an instruction to be fetched and executed is often given in **MPU cycles**. We will define an MPU cycle as the minimum time required to fetch a data byte from memory. Thus, the fetch phase of an instruction requires one MPU cycle. In inherent and immediate addressing modes, the execute phase also requires one MPU cycle. Therefore, the **minimum** time required to fetch and execute any instruction is two MPU cycles. As you will see later, other addressing modes will require more MPU cycles.

Because of their fast execution time and their efficient use of memory, the inherent and immediate modes of addressing should be used wherever possible. However, there are many situations in which these addressing modes are simply not suitable. For this reason, every microprocessor will have instructions which use other addressing modes.

Direct Addressing

Most computer operations involve an operand. To realize its full potential, the computer must be able to manipulate the operand in many different ways. Immediate addressing of an operand is most useful when the operand is a constant that is used by only one instruction in the program. In this case, the operand can be placed immediately after that instruction's opcode.

However, there are many cases in which the operand is a variable which may be operated upon by many different instructions. In these cases, the immediate addressing mode is simply not practical and a more sophisticated form of addressing is necessary.

One way of solving this problem is to use the **direct addressing mode**. In this mode, an instruction requires two bytes of memory. The first byte is the opcode of the instruction just as before. However, the second byte is **not** the operand. Instead, the second byte is the **address** of the operand. The format of the instruction as it appears in memory is shown in Figure 2-26.

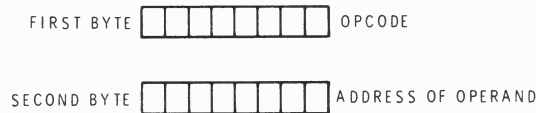


Figure 2-26

Format of an Instruction which uses
the Direct Addressing Mode.

Three typical direct-addressing-mode instructions are listed in Figure 2-27. The first is the load accumulator instruction (LDA). Read the description carefully and note the difference between this instruction and the LDA immediate instruction discussed earlier. An example of each may help. In the **immediate addressing mode**, the instruction

LDA 50_{10}

means “load 50_{10} into the accumulator.” But in the **direct addressing mode**, the same instruction means “load the number at memory location 50_{10} into the accumulator.”

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	$1001\ 0110_2$ or 96_{16}	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add	ADD	$1001\ 1011_2$ or $9B_{16}$	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator	STA	$1001\ 0111_2$ or 97_{16}	Store the contents of the accumulator in the memory location whose address is given by the next byte.

Figure 2-27

Direct Addressing Mode Instructions.

You may have noticed that this instruction has a different opcode from the LDA immediate instruction. This is necessary to tell the MPU the exact nature of the instruction. That is, the opcode tells the MPU the addressing mode as well as the operation that is to be performed.

The ADD instruction also has a slightly different meaning in the direct addressing mode. Recall that, in the **immediate** addressing mode,

ADD 10_{10}

means "add 10_{10} to the contents of the accumulator." However, in the **direct** addressing mode,

ADD 10_{10}

means "add the contents of memory location 10_{10} to the contents of the accumulator." Once again, the opcode tells the MPU the addressing mode. An opcode of $8B_{16}$ means ADD immediate whereas an opcode of $9B_{16}$ means ADD direct.

The last instruction shown is a store accumulator (STA) instruction. It tells the MPU to store the contents of the accumulator in the address indicated by the second byte of the instruction.

For example:

STA 20_{10}

means "store the contents of the accumulator in memory location 20_{10} ." Because the second byte of the instruction must be an address, there is no STA immediate instruction.

The direct addressing mode instructions require one or more additional MPU cycles to execute. A typical fetch-execute sequence is illustrated in Figure 2-28. The instruction fetch is the same regardless of the addressing mode. However, the execution phase is extended since the MPU must first obtain the address of the operand from memory and then retrieve the operand itself from memory.

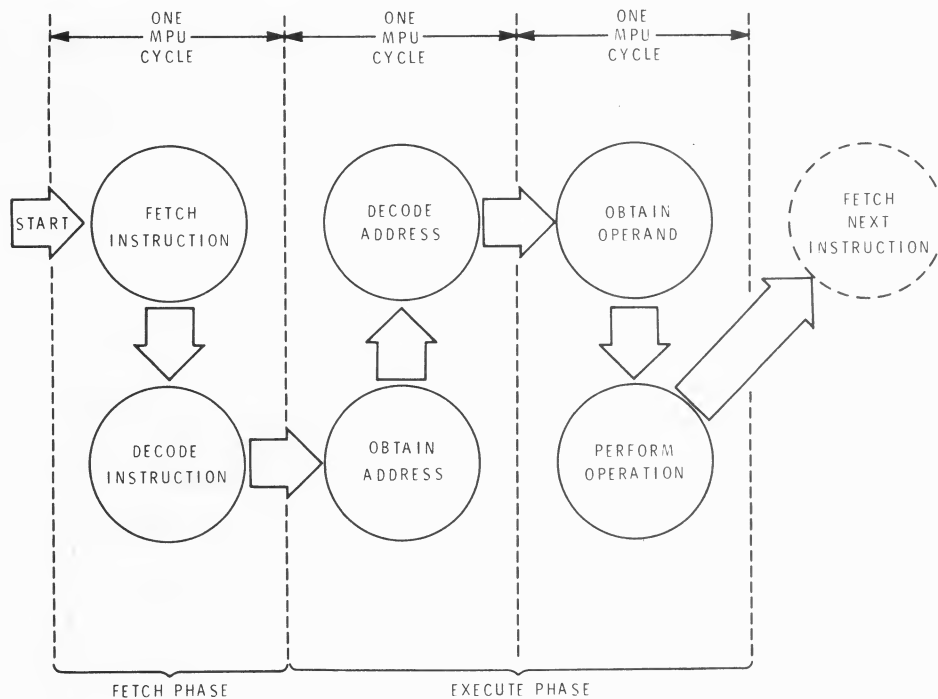


Figure 2-28
Most Direct Addressing Mode Instructions Require Three MPU Cycles.

Direct addressing generally requires more memory and longer execution times. However, there are many cases in which the added flexibility makes direct addressing worthwhile in spite of these disadvantages.

Sample Program Using Direct Addressing

The differences between direct and immediate addressing can be illustrated by a sample program. Earlier we examined a program which added two numbers (7 and 10). The sum was placed in the accumulator. Now let's look at the same program using direct addressing, only this time the sum will be stored in memory.

Figure 2-29 shows the program as it would look when stored in memory. Assume that we have arbitrarily stored the program starting at memory location or address $0001\ 0000_2$ (16_{10}). Addresses 16_{10} and 17_{10} contain the first instruction:

LDA 23_{10}

BINARY ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0000	1001 0110	LDA {
0001 0001	0001 0111	23_{10} } 1st Instruction
0001 0010	1001 1011	ADD {
0001 0011	0001 1000	24_{10} } 2nd Instruction
0001 0100	1001 0111	STA {
0001 0101	0001 1001	25_{10} } 3rd Instruction
0001 0110	0011 1110	HLT {
0001 0111	0000 0111	7_{10} } 4th Instruction
0001 1000	0000 1010	10_{10} } Data
0001 1001	0000 0000	Reserved for sum

Figure 2-29
Sample Program Using Direct Addressing.

This instruction tells the MPU to load the contents of memory location 23_{10} into the accumulator. Looking down to address 23_{10} ($0001\ 0111_2$), you see that it contains the operand 7. Thus, the first instruction causes 7 to be loaded into the accumulator.

The second instruction is in memory locations 18_{10} and 19_{10} . It is:

ADD 24_{10}

This tells the MPU to add the number at address 24_{10} to the number in the accumulator. Address 24_{10} ($0001\ 1000_2$) contains the number 10_{10} . Therefore, the second instruction causes 10_{10} to be added to the contents of the accumulator. The sum (17_{10}) is placed back in the accumulator.

The third instruction, in locations 20_{10} and 21_{10} , is:

STA 25_{10}

This tells the MPU to store the contents of the accumulator in memory location 25_{10} . After this instruction is executed, the number 17_{10} will appear in location 25_{10} .

The final instruction tells the MPU to halt. The program illustrates the value of the HLT instruction. Let's assume that the HLT instruction is inadvertently omitted. In this case, the MPU would fetch the next byte in sequence and attempt to execute it as if it were an instruction. The next byte is the number 7_{10} . This is a data word and was never intended to be an instruction. Nevertheless, the MPU probably has an instruction with an opcode of 7_{10} . After executing this instruction, the MPU will continue to fetch and execute whatever it finds in the remaining memory locations. Without a HLT instruction, it has no way of knowing where the instructions end and data begins.

Executing the Sample Program

The data flow within the microcomputer is slightly different for the direct addressing mode. Figure 2-30 shows several of the data paths within the microcomputer. Using this type of diagram, let's examine the data manipulations that occur during the execution of our sample program.

Notice that our program is loaded in memory starting at address 16_{10} . The program counter is set to 16_{10} , so the MPU is ready to begin executing the program.

The first fetch phase is illustrated in Figure 2-30. During this phase:

1. The contents of the program counter are loaded into the address register.
2. The program counter is incremented to 17_{10} .
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The contents of the data register are decoded.
6. The MPU recognizes that an LDA direct operation is indicated. This concludes the fetch phase.

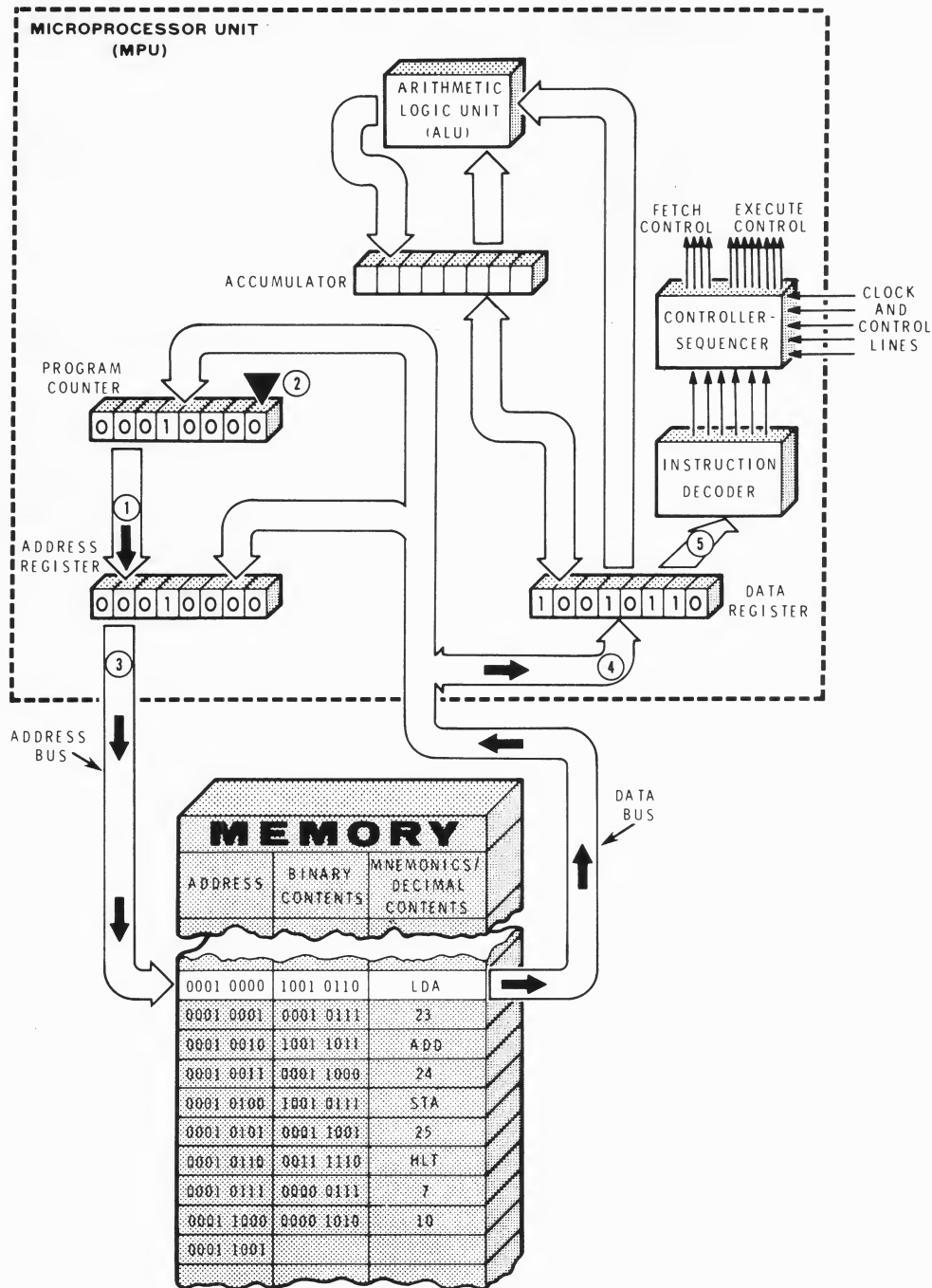


Figure 2-30
Fetching the Opcode of the First Instruction.

The execute phase has two parts when direct addressing is indicated. Figure 2-31 illustrates the first half of the execute phase. During this half:

1. The contents of the program counter are transferred to the address register. This number is the memory location that holds the address of the operand.
2. The program counter is incremented to 18_{10} .
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are placed on the data bus. However, in the direct addressing mode, this data is transferred to the address register. Thus, 23_{10} goes into the address register, replacing the previous contents. After this cycle, the address register will appear as shown in Figure 2-32.

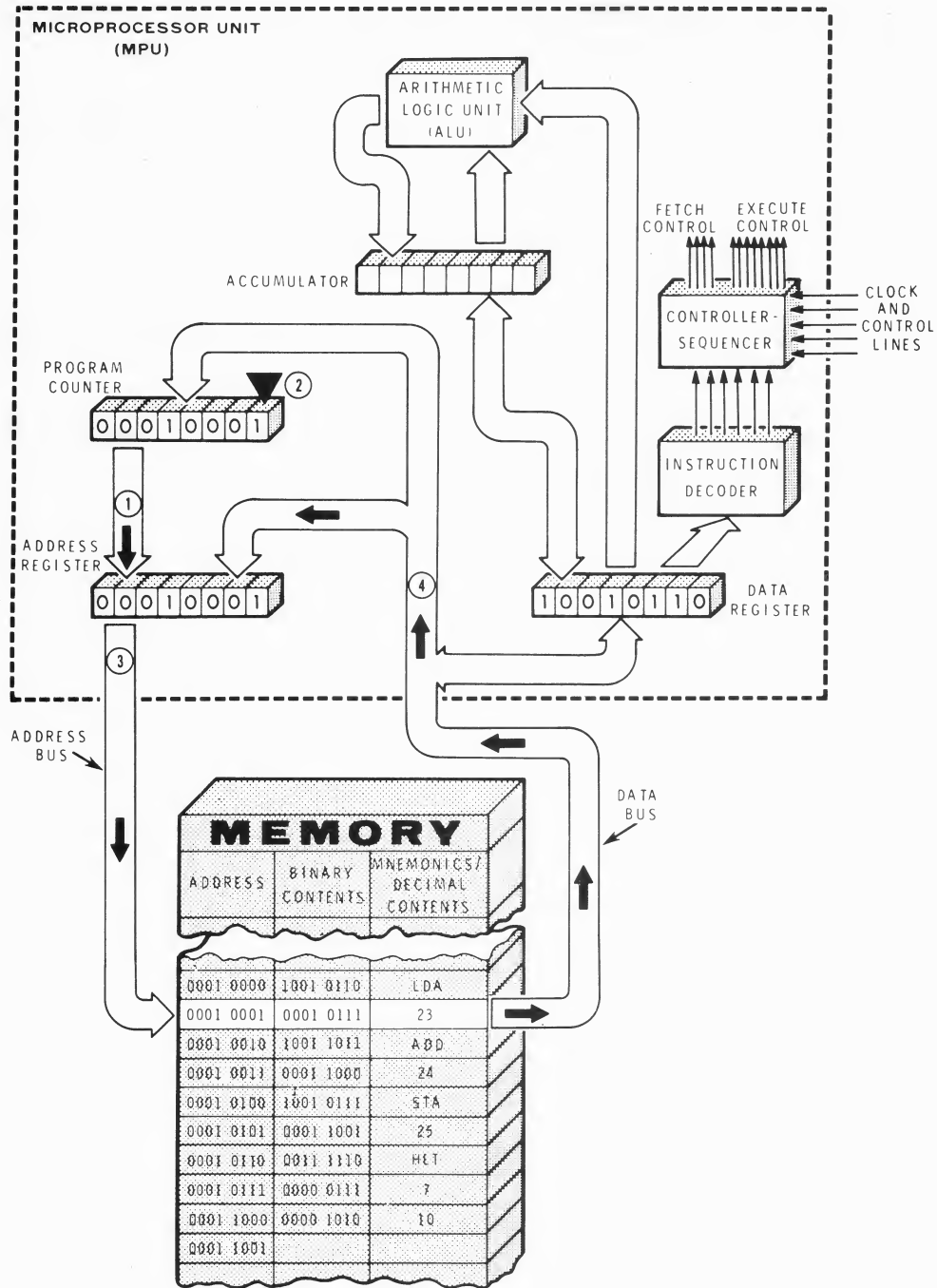


Figure 2-31
Fetching the Address of the First Operand

During the second half of the execute phase, the operand is loaded into the accumulator as shown in Figure 2-32. The procedure is:

1. The address of the operand which is in the address register is placed on the address bus.
2. The operand is read out of memory location 23_{10} and is transferred via the data bus to the data register.
3. The operand is transferred from the data register to the accumulator.

This completes the execution of the first instruction. Notice that the first operand (7_{10}) is now in the accumulator.

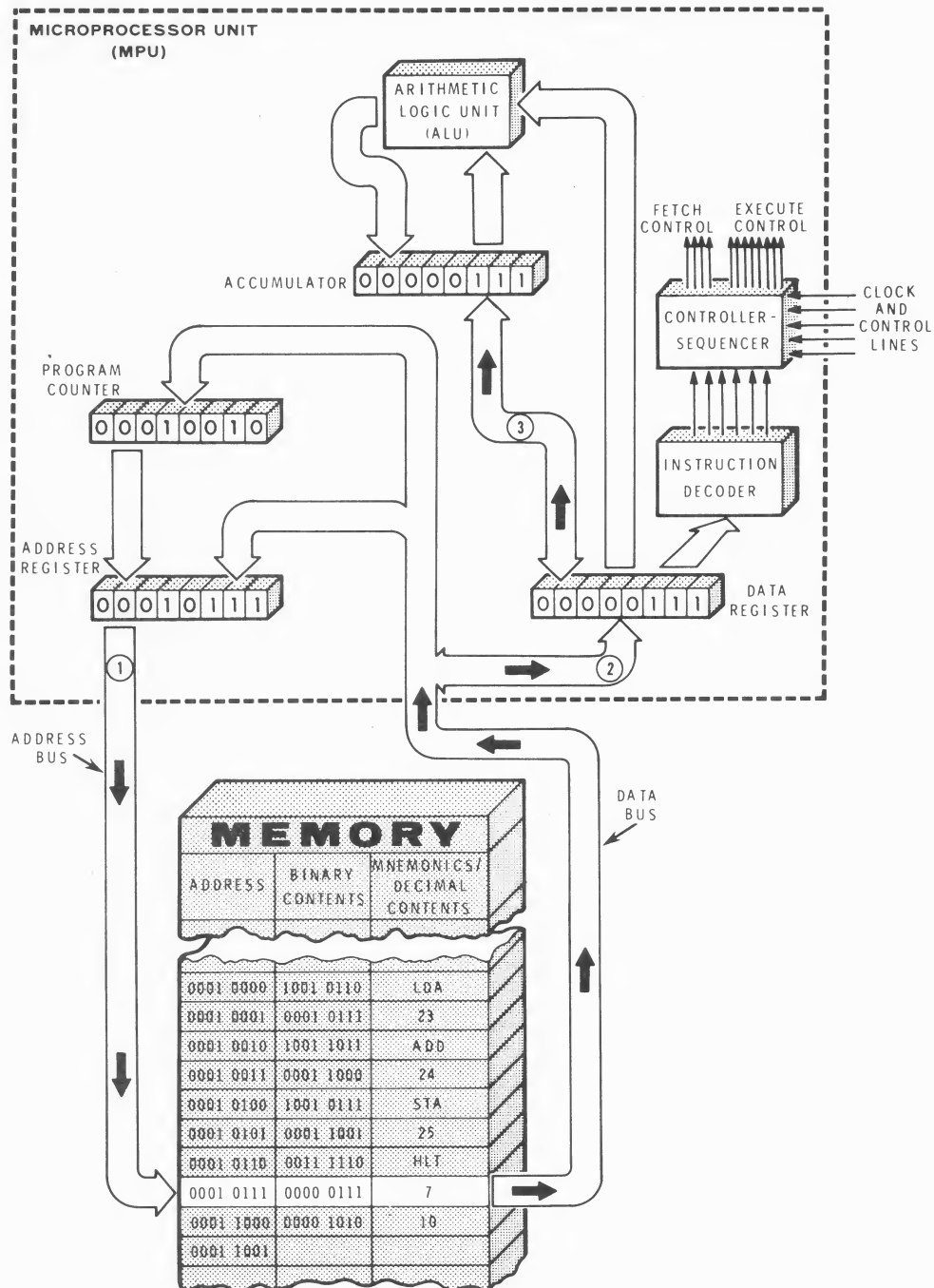


Figure 2-32
Fetching the First Operand.

The fetch phase for the second instruction is similar to that of the first. As shown in Figure 2-33, it causes the opcode of the ADD instruction to be read out of address 18₁₀. The opcode is transferred to the instruction decoder via the data bus and data register. In the process, the program counter is incremented to 19₁₀.

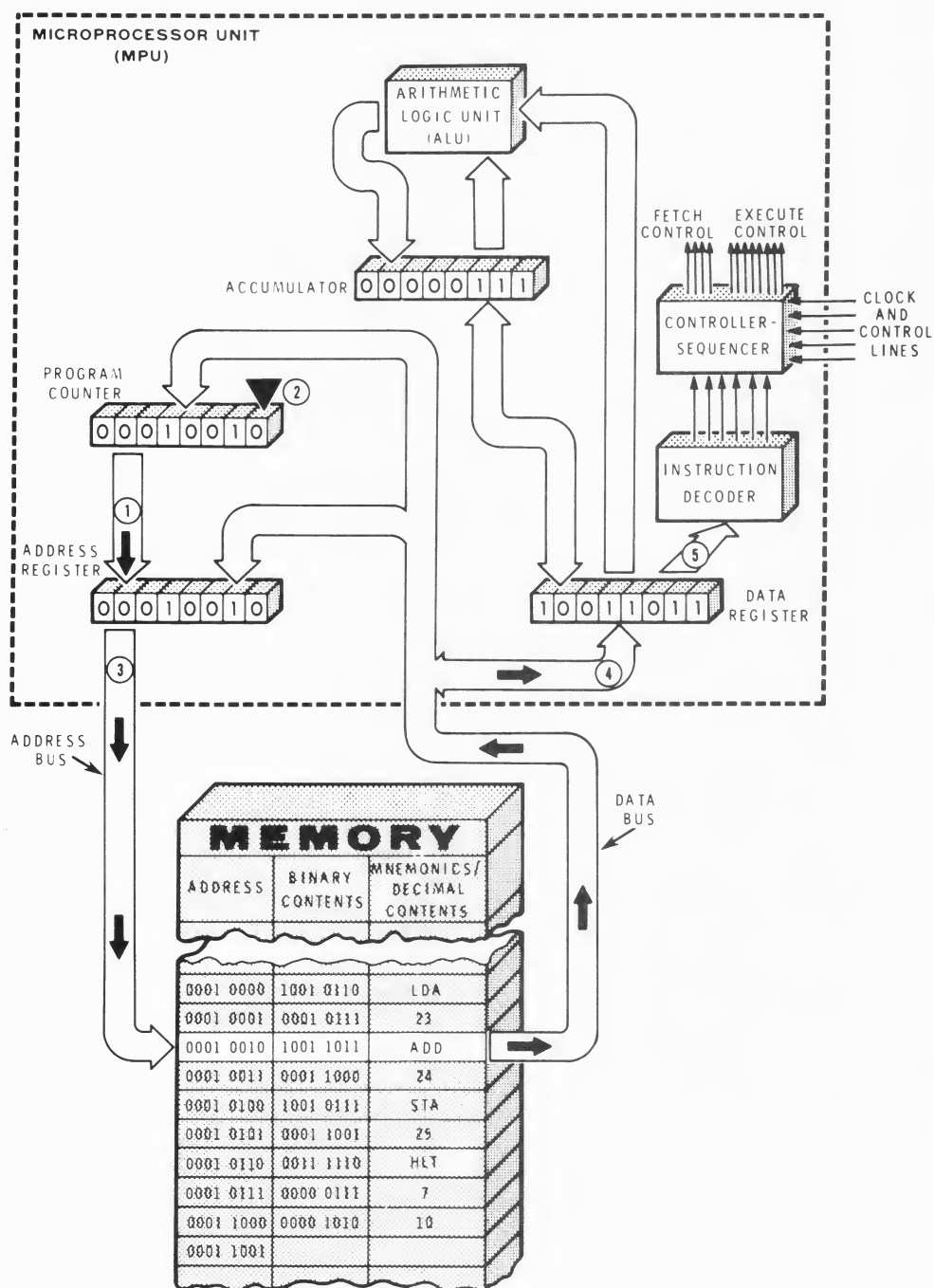


Figure 2-33
Fetching the Opcode of the Second Instruction.

The diagram illustrates the internal architecture of a Microprocessor Unit (MPU) and its connection to memory. The MPU is enclosed in a dashed box and contains several key components:

- Program Counter:** A register holding the address of the next instruction to be executed. It outputs to the Address Register.
- Address Register:** Receives addresses from the Program Counter and outputs them to the Address Bus.
- Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations on data from the Accumulator and Data Register.
- Accumulator:** A register used for storing intermediate results of calculations.
- Controller-Sequencer:** Manages the flow of execution, receiving control signals and outputting Fetch and Execute Control signals.
- Instruction Decoder:** Decodes instructions received from memory into specific control signals for the ALU and other units.
- Data Register:** Temporarily stores data being processed or moved between memory and the ALU.

The MPU interacts with external **Memory** through two buses:

- Address Bus:** Carries addresses from the MPU's internal registers to memory locations.
- Data Bus:** Carries data between the MPU's internal registers and memory.

A sample memory table is provided below:

ADDRESS	BINARY CONTENTS	MNEMONICS/DECIMAL CONTENTS
0001 0000	1001 0110	LDA
0001 0001	0001 0111	23
0001 0010	1001 1011	ADD
0001 0011	0001 1000	24
0001 0100	1001 0111	STA
0001 0101	0001 1001	25
0001 0110	0011 1110	MUL
0001 0111	0000 0111	7
0001 1000	0000 1010	10
0001 1001		

Fetching the Address of the Second Operand.

Figure 2-35 illustrates the second cycle of the execute phase. Here the address of the second operand is transferred from the address register to the address bus. The address is 24_{10} . Therefore, the contents of location 24_{10} are placed on the data bus and transferred to the data register. That is, the second operand 10_{10} is loaded into the data register. Then, the operand from the data register is made available at one input to the ALU. Simultaneously, the first operand which has been waiting in the accumulator is made available at the other input to the ALU. The ALU adds the two operands together, producing a result of 17_{10} . This sum is put back in the accumulator, replacing the previous number.

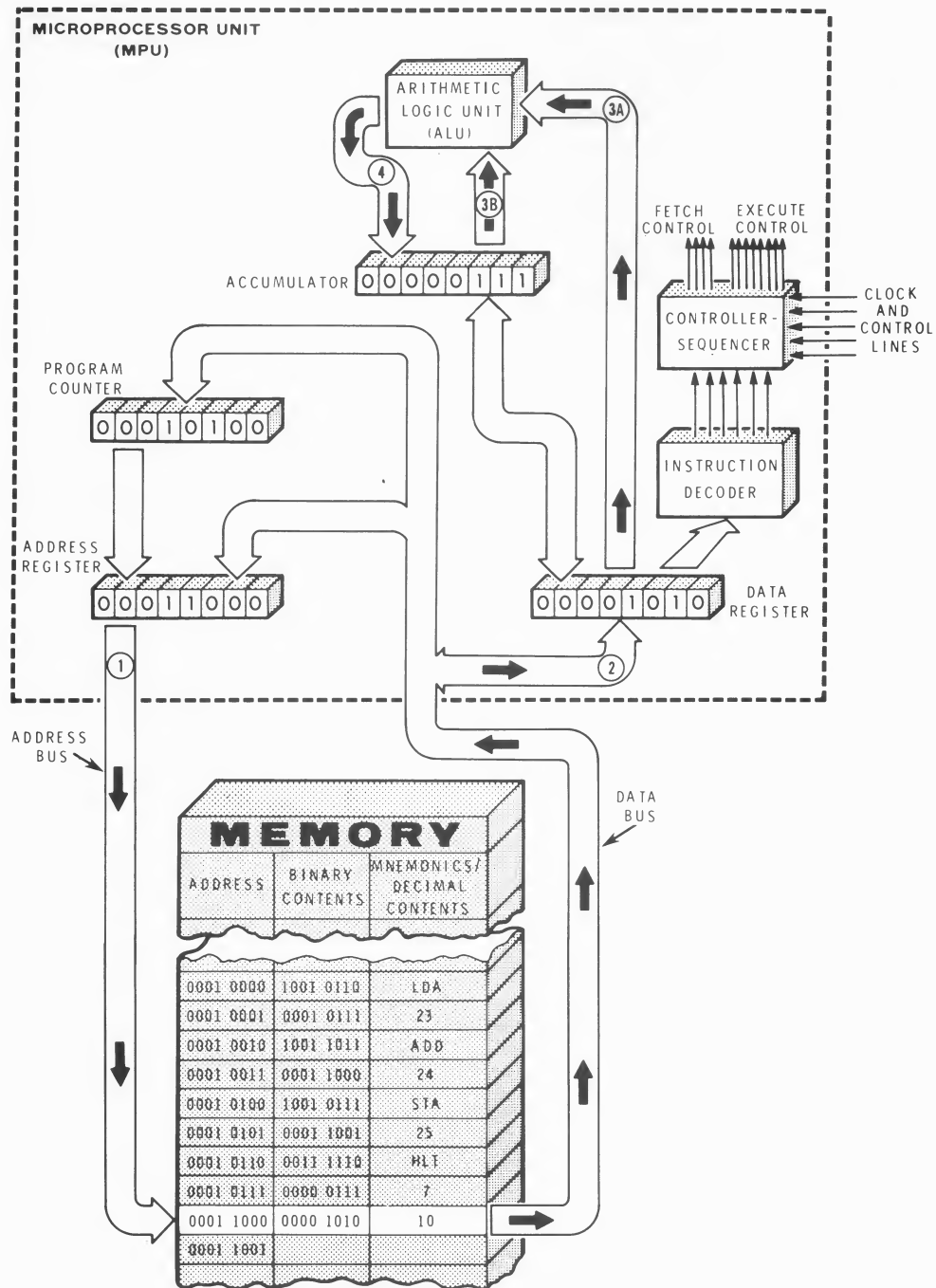


Figure 2-35
Adding the Two Operands.

Now all that remains is to place the sum in memory. This is done by the STA 25₁₀ instruction. Since this is the next instruction in sequence, it will be fetched and executed next. The fetch phase is illustrated in Figure 2-36. It ends with the STA opcode being decoded.

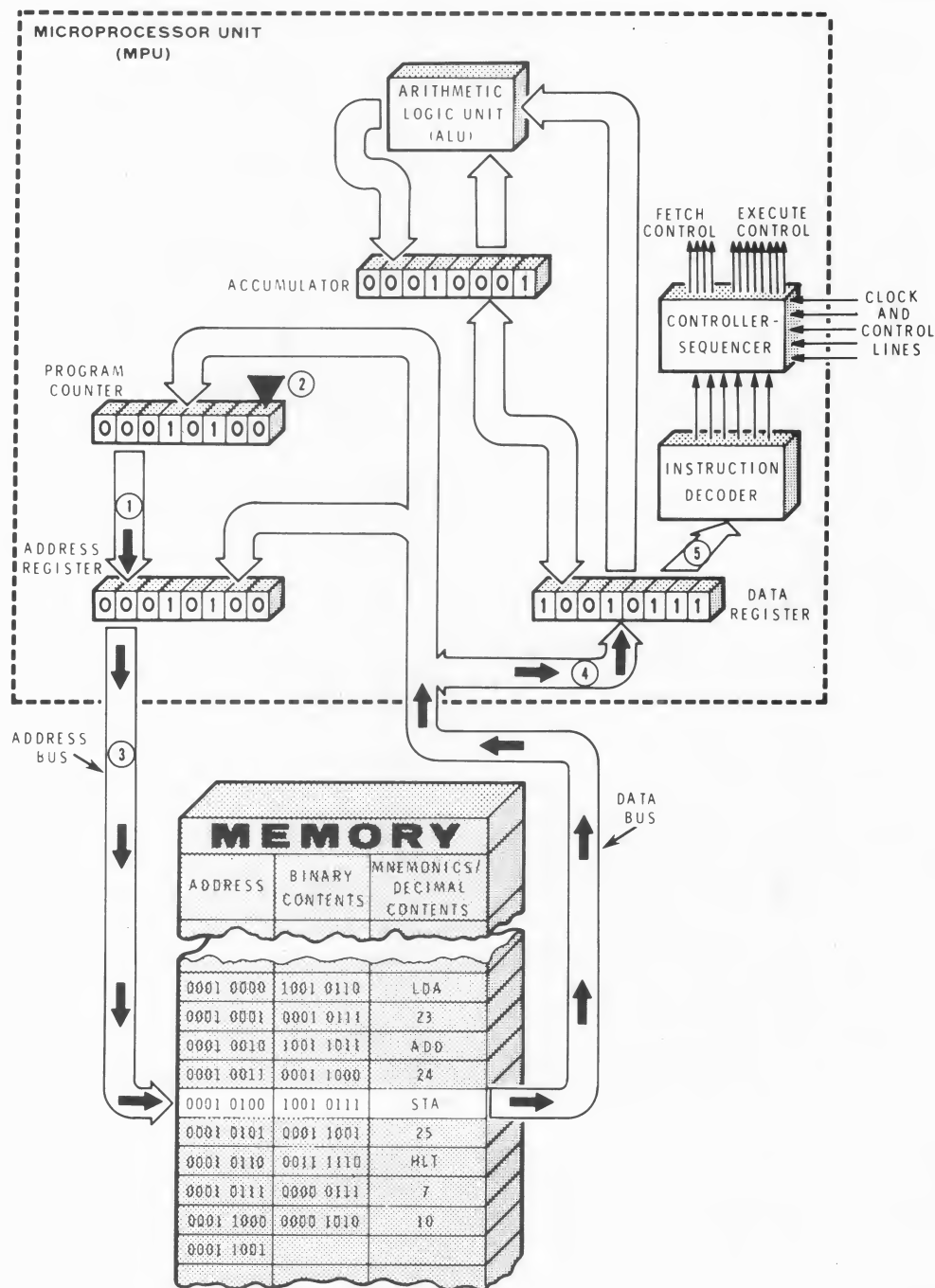


Figure 2-36
Fetching the Third Opcode.

The first half of the execution phase of the STA instruction involves loading the address of the storage location into the address register. Figure 2-37 illustrates that this four-step procedure is identical to that performed for the previous two instructions. It ends with the address 25₁₀ in the address register.

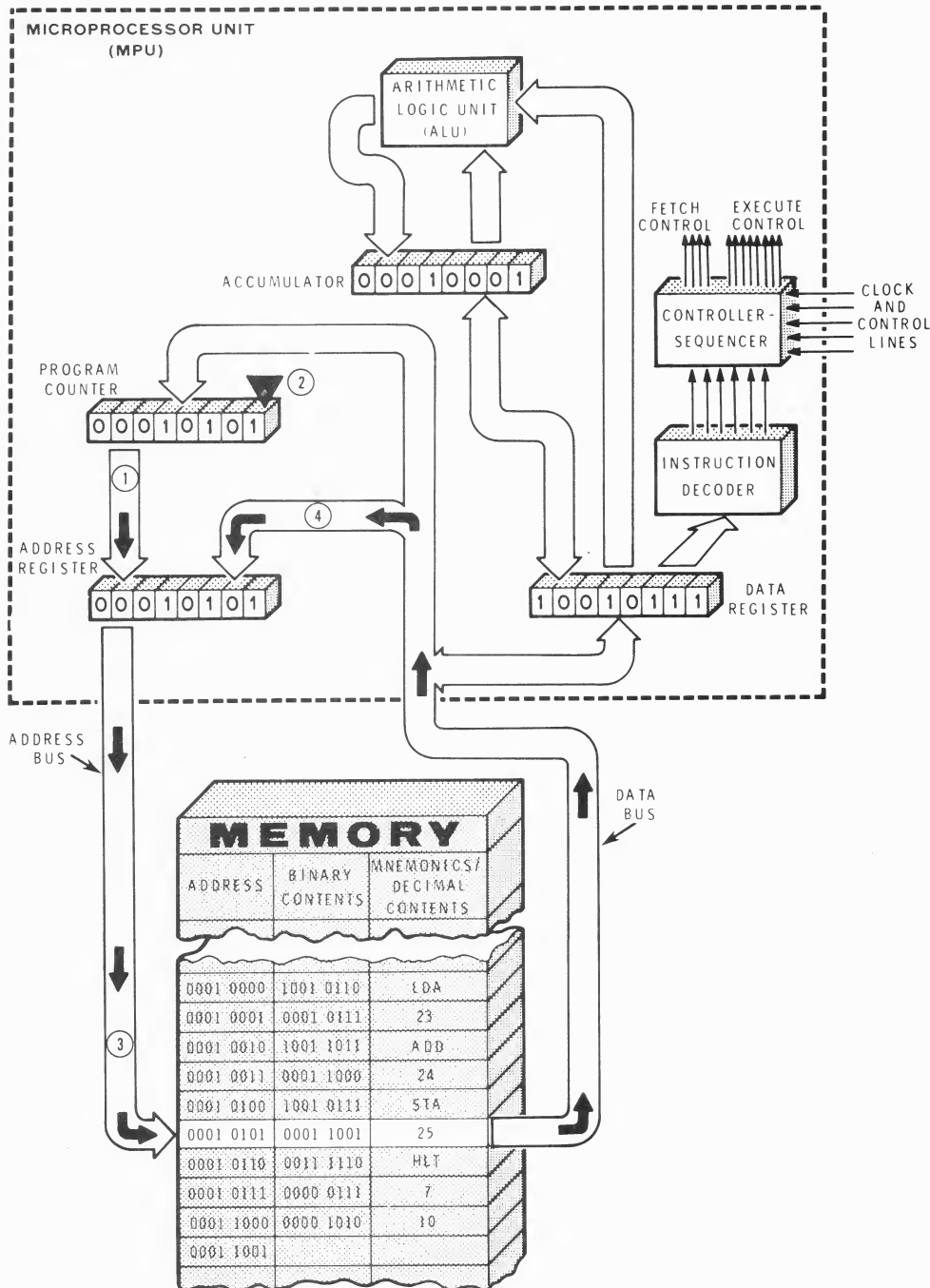


Figure 2-37
Fetching the Third Address.

During the final half of the execute phase, the contents of the accumulator are transferred to the data register and are then stored in the selected memory location. We have not yet discussed this operation in detail. Therefore, the step-by-step procedure is presented below. Refer to Figure 2-38 for the following steps:

1. The contents of the accumulator (17_{10}) are transferred to the data register. At this point, the number 17_{10} exists in both the accumulator and the data register.
2. The address at which this data is to be stored is placed on the address bus.
3. The contents of the data register are placed on the data bus.
4. The number on the data bus is written into the selected memory location. That is, 17_{10} is written into memory location 25_{10} .

Notice that, after this operation, the number 17_{10} appears at memory location 25_{10} , but it also appears in the accumulator. Thus, the number is merely "copied" into memory. It is also important to note that the previous contents of memory location 25_{10} are lost whenever you write new data into this location. For this reason, you must be certain that you do not write into a location that contains an instruction or some byte of data that you will need later.

The program has now accomplished its goal. It has added 10 to 7 and has stored the sum back in memory. The last step in the program is the HLT instruction. The MPU fetches and executes this instruction next. The fetch and execute sequence for this instruction were discussed earlier and need not be repeated here.

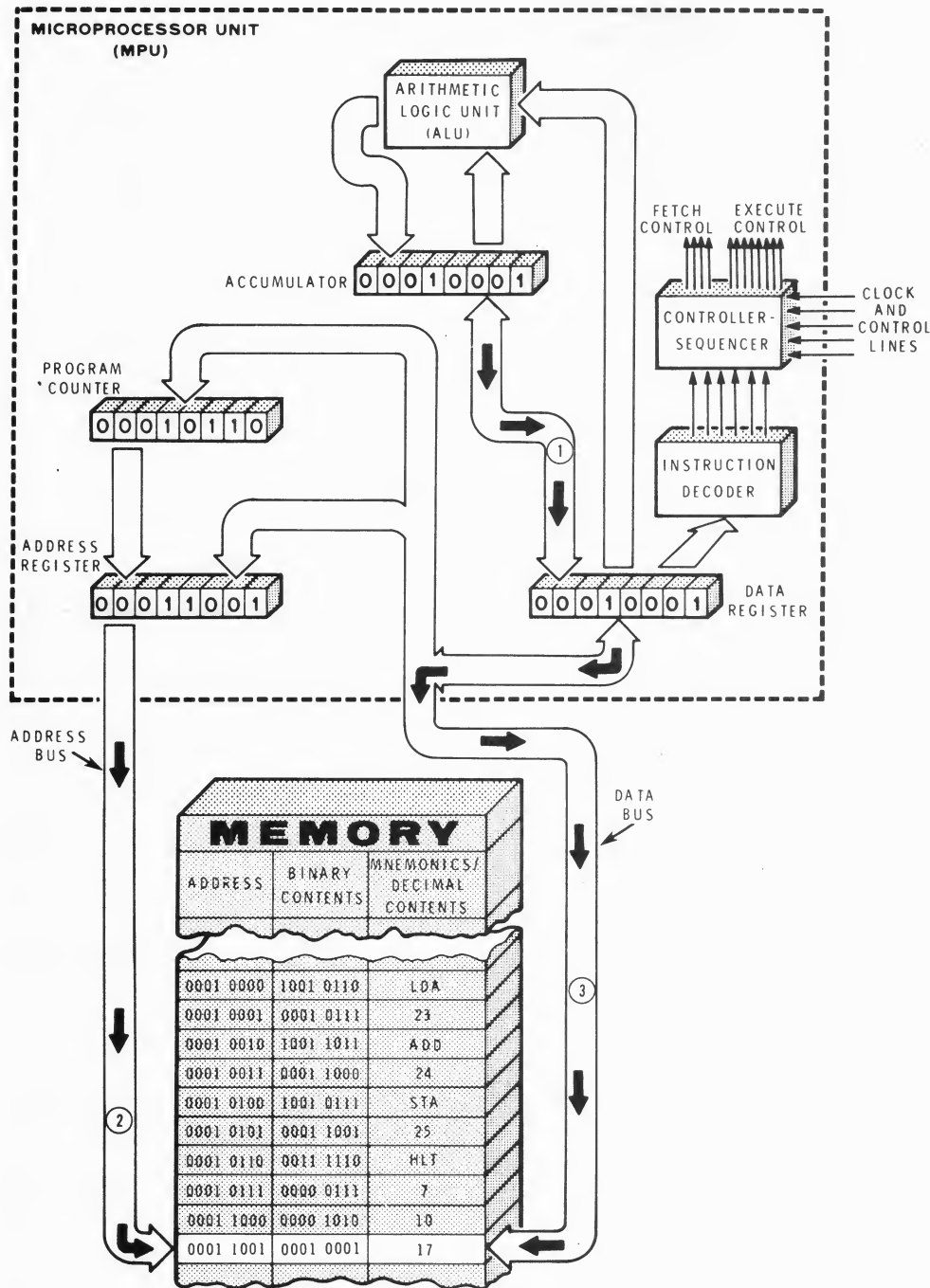


Figure 2-38
Storing the Sum.

Combining Addressing Modes

When writing programs, you can use the addressing mode that best suits your application. For example, the program that was just discussed can be shortened by using the immediate addressing mode with the first two instructions. Figure 2-39 compares two programs that do the same job.

Using direct addressing only, the program required ten bytes of memory. Its execution requires eleven MPU cycles. If you use immediate addressing for the first two instructions, the program requires eight bytes of memory. Furthermore, it can be executed in nine MPU cycles. Everything else being equal, the second approach would probably be preferred.

A. USING DIRECT ADDRESSING

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with operand 1 which is stored at this address. Add to accumulator direct with operand 2 which is stored at this address. Store the sum at this address. Stop Operand 1 Operand 2 Reserved for sum.
01	07	07 ₁₀	
02	9B	ADD	
03	08	08 ₁₀	
04	97	STA	
05	09	09 ₁₀	
06	3E	HLT	
07	21	33 ₁₀	
08	17	23 ₁₀	
09	—	—	

B. COMBINING ADDRESSING MODES

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
00	86	LDA	Load accumulator immediately with Operand 1. Add to accumulator immediately with Operand 2. Store the sum at this address. Stop Reserved for sum.
01	21	33 ₁₀	
02	8B	ADD	
03	17	23 ₁₀	
04	97	STA	
05	07	07 ₁₀	
06	3E	HLT	
07	—	—	

Figure 2-39
By Combining the Addressing Modes,
We Can Save Memory Space And
Computer Time.

Self-Test Review

41. What addressing mode is used by single byte instructions?
42. In the immediate addressing mode, what is the second byte of the instruction?
43. In the direct addressing mode, what is the second byte of the instruction?
44. In all addressing modes, what is the first byte of the instruction?
45. Define MPU cycle.
46. Which of the three addressing modes discussed so far requires the longest execution time?
47. Refer to Figure 2-39A. What number is loaded into the accumulator by the first instruction?
48. What number is added to the accumulator by the second instruction?
49. When the computer halts, what number will be in memory location 09?
50. Refer to Figure 2-39B. When the computer halts, what number will be in memory location 07?

Answers

- 41. Inherent or implied.
- 42. The operand.
- 43. The address of the operand.
- 44. The opcode.
- 45. An MPU cycle is the time required to fetch a byte from memory.
- 46. The direct addressing mode.
- 47. 21_{16} or 33_{10} .
- 48. 17_{16} or 23_{10} .
- 49. 38_{16} or 56_{10} .
- 50. 38_{16} or 56_{10} .

EXPERIMENT 3

Perform Experiment 3 in Unit 7 of this course. After you finish the experiment, return to this unit and complete the final examination.

UNIT EXAMINATION

1. In microprocessor terminology, the number or piece of data that is operated upon is called the:
 - A. Operand.
 - B. Opcode.
 - C. Address.
 - D. Instruction.

2. The part of the instruction that tells the microprocessor what operation to perform is called the:
 - A. Operand.
 - B. Opcode.
 - C. Address.
 - D. Mnemonic.

3. The portion of the microcomputer in which instructions and data are stored is called the:
 - A. ALU.
 - B. MPU.
 - C. RAM.
 - D. Data bus.

4. An 8-bit byte in memory can represent an:
 - A. Opcode.
 - B. Operand.
 - C. Address.
 - D. All of the above.

5. During the fetch phase:
 - A. The opcode is fetched from memory and is decoded.
 - B. The address of the operand is fetched from memory and is decoded.
 - C. The operand is fetched from memory and is operated upon.
 - D. The program count is fetched from memory.

6. In what register is the result of an arithmetic operation normally placed?
 - A. The data register.
 - B. The address register.
 - C. The arithmetic logic unit (ALU).
 - D. The accumulator.
7. During the fetch and execute phases of the "load accumulator direct" instruction, the information on the data bus will be:
 - A. The operand address followed by the operand.
 - B. The program count, followed by the opcode, followed by the operand address, followed by the operand.
 - C. The opcode, followed by the operand address, followed by the operand.
 - D. The opcode, followed by the operand.
8. In the immediate addressing mode, the second byte of the instruction is the:
 - A. Opcode of the instruction.
 - B. Number that is to be operated upon.
 - C. Address of the operand.
 - D. Address of the opcode.
9. In the direct addressing mode, the second byte of the instruction is the:
 - A. Opcode of the instruction.
 - B. Number that is to be operated upon.
 - C. Address of the operand.
 - D. Address of the opcode.
10. Which of the following is normally a one-byte instruction?
 - A. Halt.
 - B. Add immediate.
 - C. Load accumulator direct.
 - D. Store accumulator direct.

11. At the start of the fetch phase, the program counter contains:
- A. The address of the operand to be fetched.
 - B. The address of the opcode to be fetched.
 - C. The opcode of the instruction.
 - D. The operand.
12. Which register holds the opcode while it is being decoded?
- A. The address register.
 - B. The accumulator.
 - C. The data register.
 - D. The program counter.
13. The program shown in Figure 2-40:

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	86	LDA
01	00	00 ₁₆
02	97	STA
03	09	09 ₁₆
04	97	STA
05	0A	0A ₁₆
06	97	STA
07	0B	0B ₁₆
08	3E	HLT
09	—	—
0A	—	—
0B	—	—

Figure 2-40
Program for Question 13.

- A. Adds the contents of memory location 09, 0A, and 0B.
- B. Stores 00 in locations 09, 0A, 0B.
- C. Stores 09 in location 03, 0A in location 05, and 0B in location 07.
- D. Stores 0B in the accumulator.

14. The program shown in Figure 2-41:

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	96	LDA
01	09	09 ₁₆
02	9B	ADD
03	09	09 ₁₆
04	9B	ADD
05	09	09 ₁₆
06	9B	ADD
07	09	09 ₁₆
08	3E	HLT
09	04	04 ₁₆

Figure 2-41

Program for Question 14.

- A. Multiplies 4 times 4 and holds the product in the accumulator.
- B. Multiplies 9 times 3 and holds the product in the accumulator.
- C. Multiplies 4 times 3 and stores the product in the accumulator.
- D. Multiplies 9 times 4 and holds the product in the accumulator.

15. The program shown in Figure 2-42:
- A. Swaps the contents of memory location 0D and 0E.
 - B. Stores AA_{16} in locations 0D, 0E, and 0F.
 - C. Stores BB_{16} in locations 0D, 0E, and 0F.
 - D. Adds AA and BB, storing the sum at location 0F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	96	LDA
01	0D	$0D_{16}$
02	97	STA
03	0F	$0F_{16}$
04	96	LDA
05	0E	$0E_{16}$
06	97	STA
07	0D	$0D_{16}$
08	96	LDA
09	0F	$0F_{16}$
0A	97	STA
0B	0E	$0E_{16}$
0C	3E	HLT
0D	AA	AA_{16}
0E	BB	BB_{16}
0F	—	—

Figure 2-42
Program for Question 15.

EXAMINATION ANSWERS

1. A — The number or piece of data that is operated upon by the microprocessor is called the operand.
2. B — The part of the instruction that tells the microprocessor what operation to perform is called the opcode.
3. C — The instructions and data are stored in memory. RAM stands for random access read/write memory.
4. D — An 8-bit byte in memory can represent an opcode, an operand, or an address.
5. A — During the fetch phase, the opcode is fetched from memory and is decoded.
6. D — The result of an arithmetic operation is normally stored in the accumulator.
7. C — Recall that the opcode appears on the data bus first. Next, the address of the operand is fetched. It appears on the data bus second. Finally, the operand itself is fetched. It appears on the data bus third.
8. B — In the immediate addressing mode, the second byte of the instruction is the operand.
9. C — In the direct addressing mode, the second byte of the instruction is the address of the operand.
10. A — The “halt” instruction is a single byte instruction.

11. B — At the start of the fetch phase, the program counter contains the address of the next opcode to be fetched.
12. C — The data register holds the opcode while it is being decoded.
13. B — The first instruction is "load accumulator immediate." It loads 00 into the accumulator. The next three instructions store the contents of the accumulator (00) at locations 09, 0A, and 0B respectively.
14. A — The first instruction loads the operand at address 09 into the accumulator. Notice that the operand is 04. The next three instructions successively add 04 to the accumulator. In effect, the program multiplies 4 times 4. It does this by successive addition.
15. A — The program swaps the contents of memory locations 0D and 0E. In the process it used location 0F as a temporary storage place for operand AA.

Unit 3

COMPUTER ARITHMETIC

CONTENTS

Introduction	3-3
Unit Objectives	3-4
Unit Activity Guide	3-5
Binary Arithmetic	3-6
Two's Complement Arithmetic	3-26
Boolean Operations	3-35
Experiment 4	3-44
Unit Examination	3-45
Examination Answers	3-47

INTRODUCTION

In this Unit you will complete your study of the binary number system. Since microprocessors use binary numbers for data and control, it is important that you become familiar with them.

Computer arithmetic involves many forms of number manipulation. In the pages that follow you will be given the fundamentals of binary mathematics: addition, subtraction, multiplication, and division. Then you will learn to perform two's complement arithmetic using binary numbers. Finally, you will be shown how the microprocessor performs the four basic Boolean logic operations. These logical operations include AND, OR, exclusive OR, and invert.

UNIT OBJECTIVES

When you complete this Unit you will be able to:

1. Add two binary numbers.
2. Subtract one binary number from another.
3. Multiply one binary number by another.
4. Divide one binary number by another.
5. Derive the one's complement of a binary number.
6. Derive the two's complement of a binary number.
7. Add binary numbers using two's complement arithmetic.
8. Manipulate binary numbers using the AND operation.
9. Manipulate binary numbers using the OR operation.
10. Manipulate binary numbers using the exclusive OR operation.
11. Logically invert binary numbers.

UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on Binary Arithmetic.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 1-11.	_____
<input type="checkbox"/> Read the section on Two's Complement Arithmetic.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 12-21.	_____
<input type="checkbox"/> Read the section on Boolean Operations.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 22-30.	_____
<input type="checkbox"/> Perform Experiment 4.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Review the Examination Answers.	_____

BINARY ARITHMETIC

A number system can be used to perform two basic operations: addition and subtraction. But by using addition and subtraction, you can then perform multiplication, division, and any other numerical operation. In this section, binary arithmetic (addition, subtraction, multiplication, and division) will be examined, using decimal arithmetic as a guide.

Binary Addition

Binary addition is performed somewhat like decimal addition. If two decimal numbers, 56719 and 31863, are added together, the sum 88582 is obtained. You could analyze the details of this operation in the following manner.

NOTE: In the following explanations, the term "first column" refers to the first column of figures you work with in the problem — the column on the right (9, 3, and 2 in the following example). The term "second column" refers to the second column you work with, etc.

Carry:	00101
Addend:	56719
Augend:	+ 31863
Sum:	<u>88582</u>

Adding the first column, decimal numbers 9 and 3, gives the sum of 12. This is expressed in the sum as the digit 2 with a carry of 1. The carry is then added to the next column. Adding the second column decimal numbers 1 and 6, and the carry from the first column, gives the sum of 8, with no carry. This process continues until all of the columns (including carries) have been added. The sum represents the numeric value of the addend and augend. (The **addend** is the number to be added to another number, while the **augend** is the number to which the addend is added.)

When you add two binary numbers, you perform the same operation. Figure 3-1 summarizes the four rules of addition with binary numbers.

1. $0 + 0 = 0$
2. $0 + 1 = 1$
3. $1 + 1 = 0$ with a carry of 1.
4. $1 + 1 + 1 = 1$ with a carry of 1.

Figure 3-1
Rules for binary addition.

To illustrate the process of binary addition, let's add 1101 to 1101.

Carry:	1101
Addend:	1101
Augend:	<u>+ 1101</u>
Sum:	11010

In the first column, 1 plus 1 equals 0 with a carry of 1 to the second column. This agrees with rule 3. In the second column, 0 plus 0 equals 0 with no carry. To this sum, the carry from the first column is added. Thus, 0 plus 1 equals 1 with no carry. These two additions in the second column give a total sum of 1 with a carry of 0. Rules 1 and 2 were used to obtain the sum.

In column three, 1 plus 1 equals 0 with a carry of 1. To this sum, the second column carry is added. This yields a third column sum of 0 with a carry of 1 to column four. Rules 3 and 1 were used to obtain the sum.

In column four, 1 plus 1 equals 0 with a carry of 1. To this sum, the third column carry is added. This yields a fourth column sum of 1 with a carry to the fifth column. Rule 4 allows you to add three binary 1's and obtain 1 with a carry of 1.

In column five, there is no addend or augend. Therefore, you can assume rule 2 and add the carry to 0 to obtain the sum of 1. Thus, the sum of 1101_2 plus 1101_2 equals 11010_2 . You can verify this by converting the binary numbers to decimal numbers.

Now study the following two examples of binary addition, where 10001111_2 is added to 10110101_2 and 111011_2 is added to 11001100_2 .

$$\begin{array}{r}
 \text{Carry:} \quad 10111111 \\
 \text{Addend:} \quad 10110101 \\
 \text{Augend:} \quad + 10001111 \\
 \hline
 \text{Sum:} \quad 101000100
 \end{array}$$

$$\begin{array}{r}
 \text{Carry:} \quad 11111000 \\
 \text{Addend:} \quad 11001100 \\
 \text{Augend:} \quad + 00111011 \\
 \hline
 \text{Sum:} \quad 100000111
 \end{array}$$

When binary addition is performed with a microprocessor, 8-bit numbers are generally used. As shown in the last example, two zeros were added after the MSB of the augend to produce an 8-bit number. After addition, a 1 in the ninth bit is represented as the "carry" bit by the microprocessor. This will be explained in a later unit.

Binary Subtraction

Binary subtraction is performed exactly like decimal subtraction. Therefore, before binary subtraction can be attempted, decimal subtraction should be reexamined. You know that if decimal 5486 is subtracted from 8303, the difference 2817 is obtained.

$$\begin{array}{r}
 \text{Minuend after borrow:} \quad 7 \ 12 \ 9 \ 13 \\
 \text{Minuend:} \quad 8 \ 3 \ 0 \ 3 \\
 \text{Subtrahend:} \quad -5 \ 4 \ 8 \ 6 \\
 \hline
 \text{Difference:} \quad 2 \ 8 \ 1 \ 7
 \end{array}$$

Because the digit 6 in the subtrahend is larger than the digit 3 in the minuend, a 1 is borrowed from the next higher-order digit in the minuend. If that digit is 0, as in this example, 1 is borrowed from the next higher-order digit that contains a number other than 0. That digit is reduced by 1 (from 3 to 2 in this example) and the digits skipped in the minuend are given the value 9. This is equivalent to removing 1 from 30 with the result of 29, as in this example. In the decimal system, the digit borrowed has the value of ten. Therefore, the minuend digit now has the value 13, and 6 from 13 equals 7.

In the second column, 8 from 9 equals 1. Since the subtrahend is larger than the minuend in the third column, 1 is borrowed from the next higher-order digit. This raises the minuend value from 2 to 12, and 4 from 12 equals 8. In the fourth column, the minuend was reduced from 8 to 7 because of the previous borrow, and 5 from 7 equals 2.

Whenever 1 is borrowed from a higher-order digit, the borrow is equal in value to the radix or base of the number system. Therefore, a borrow in the decimal number system equals ten, while a borrow in the binary number system equals two.

When you subtract one binary number from another, you use the same method described for decimal subtraction. Figure 3-2 summarizes the four rules for binary subtraction.

1. $0 - 0 = 0$
2. $1 - 1 = 0$
3. $1 - 0 = 1$
4. $0 - 1 = 1$ with a borrow of 1.

Figure 3-2
Rules for binary subtraction.

To illustrate the process of binary subtraction, let's subtract 1101 from 11011.

Minuend after borrow:	0 10 10 1 1
Minuend:	1 1 0 1 1
Subtrahend:	<u>− 1 1 0 1</u>
Difference:	1 1 1 0

The "minuend after borrow" now shows the value of each minuend digit after a borrow occurs. Remember that binary 10 equals decimal 2.

In the first column, 1 from 1 equals 0 (rule 2). Then, 0 from 1 in the second column equals 1 (rule 3). In the third column, 1 from 0 requires a borrow from the fourth column. Thus, 1 from 10₂ equals 1 (rule 4). The minuend in the fourth column is now 0, from the previous borrow. Therefore, a borrow is required from the fifth column, so that 1 from 10₂ in the fourth column equals 1 (rule 4). Because of the previous borrow, the minuend in

the fifth column is now 0 and the subtrahend is 0 (nonexistent), so that 0 from 0 equals 0 (rule 1). The 0 in the fifth column is not shown in the difference because it is not a significant bit. Thus, the difference between 11011_2 and 1101_2 is 1110_2 . You can verify this by converting the binary numbers to decimal numbers.

As a further example of binary subtraction, subtract 00100101_2 from 11000100_2 , as shown below. Then proceed to the next example and subtract 10111010_2 from 11101110_2 .

Minuend after borrow:	1 0 1 1 1 10 1 10
Minuend:	1 1 0 0 0 1 0 0
Subtrahend:	<u>0 0 1 0 0 1 0 1</u>
Difference:	1 0 0 1 1 1 1 1

Minuend after borrow:	0 0 10 10 1 1 1 0
Minuend:	1 1 1 0 1 1 1 0
Subtrahend:	<u>1 0 1 1 1 0 1 0</u>
Difference:	0 0 1 1 0 1 0 0

When a borrow is required in the minuend, 1 is obtained from the next high-order bit that contains a 1. That bit then becomes 0, and all bits skipped (0 value bits) are given the value 1. This is equivalent to removing 1 from 1000_2 with the result of 0111_2 .

As with binary addition, microprocessors generally perform subtraction on 8-bit number groups. In the previous example, the answer contained only six significant bits, but two 0 bits were added to maintain the 8-bit grouping. This would also be true for the minuend and subtrahend.

Subtraction of a large number from a smaller number will be described in a later section of this Unit.

Binary Multiplication

Multiplication is a short method of adding a number to itself as many times as it is specified by the multiplier. However, if you were to multiply 324_{10} by 223_{10} , you would probably use the following method.

Multiplicand:	324
Multiplier:	<u>$\times 223$</u>
First partial product:	972
Second partial product:	648
Third partial product:	<u>648</u>
Carry:	<u>0121</u>
Final product:	72252

Using this short form of multiplication, you multiply the multiplicand by each digit of the multiplier and then sum the partial products to obtain the final product. Note that, for convenience, the additive carries are set-down under the partial products rather than over them as in normal addition.

Binary multiplication follows the same general principles as decimal multiplication. However, with only two possible multiplier bits (1 or 0), binary multiplication is a much simpler process. Figure 3-3 lists the rules of binary multiplication. These rules are used to multiply 1111_2 by 1101_2 on the next page.

1. $0 \times 0 = 0$

2. $0 \times 1 = 0$

3. $1 \times 0 = 0$

4. $1 \times 1 = 1$

Figure 3-3

Rules for binary multiplication.

Multiplicand:	1111
Multiplier:	<u>×1101</u>
First partial product:	1111
Second partial product:	<u>0000</u>
Carry:	<u>0000</u>
Sum of partial products:	1111
Third partial product:	<u>1111</u>
Carry:	<u>111100</u>
Sum of partial products:	1001011
Fourth partial product:	<u>1111</u>
Carry:	<u>1111000</u>
Final product:	11000011

As with decimal multiplication, you multiply the multiplicand by each bit in the multiplier and add the partial sums. First you multiply 1111_2 by the least significant multiplier bit (1) and set down the partial product so the least significant bit (LSB) is under the multiplier bit. Then you multiply the multiplicand by the next multiplier bit (0) and set down the partial product so the LSB is under the multiplier bit. Now that there are two partial products, they should be added. Although it is possible to add more than two binary numbers, keeping track of the multiple carries may become confusing. Therefore, for these examples, add only two partial products at a time.

Notice that the first partial product is identical to the multiplicand. The second partial product is all zeros. Since the binary number system contains only ones and zeros, the partial product will always equal either the multiplicand or zero. Because of this, you can obtain the third partial product by copying the multiplicand. Begin with the LSB under the third multiplier bit. Add this value to the previous partial sum. Now obtain the fourth partial product by copying the multiplicand. Begin with the LSB under the fourth multiplier bit. Add this value to the previous partial sum. This is the final product. You can verify the result by converting the binary numbers to decimal.

Reexamine the illustration for the previous multiplication example. Notice that binary multiplication is a process of shift and add. For each 1 bit in the multiplier you copy down the multiplicand, beginning with the LSB under the bit. You can ignore any zeros in the multiplier. But do not make the mistake of setting down the multiplicand under the 0 bit.

To make sure you fully understand binary multiplication, multiply 1001_2 by 1100_2 and then multiply 1101_2 by 1111_2 .

Multiplicand:	1001
Multiplier:	$\times 1100$
First partial product:	0000
Second partial product:	0000
Carry:	0000
Sum of partial products:	00000
Third partial product:	1001
Carry:	00000
Sum of partial products:	100100
Fourth partial product:	1001
Carry:	000000
Final product:	1101100

```

      1001
    × 1100
    -----
      0000
      0000
      1001
      1001
    -----
    100100
      1001
    -----
    11000011
  
```

Multiplicand:	1101
Multiplier:	$\times 1111$
First partial product:	1101
Second partial product:	1101
Carry:	11000
Sum of partial products:	100111
Third partial product:	1101
Carry:	100100
Sum of partial products:	1011011
Fourth partial product:	1101
Carry:	1111000
Final product:	11000011

```

      1101
    × 1111
    -----
      1101
      1101
      1101
      1101
    -----
    100111
      1101
    -----
    1011011
      1101
    -----
    11000011
  
```

In the first of these last two examples, the two zeros in the multiplier were included in the multiplication process. This was to insure that the multiplicand was copied down under the proper multiplier bits. The multiplication process could have been represented in this manner:

Multiplicand:	1001
Multiplier:	$\times 1100$
Third partial product:	100100
Fourth partial product:	1001
Carry:	000000
Final product:	1101100

Remember, just as in decimal multiplication, you must keep track of any zeros by setting a zero in the product under the 0 bit in the multiplier. This is very important when the zero occupies the LSB.

Binary Division

Division is the reverse of multiplication. Therefore, it is a procedure for determining how many times one number can be subtracted from another. The process you are probably familiar with is called “long” division. If you were to divide decimal 181 by 45, you would obtain the quotient, $4\frac{1}{45}$, as follows:

		004	Quotient
Divisor	45	$\overline{)181}$	Dividend
		180	
		$\underline{1}$	Remainder

Using long division, you would examine the most significant digit in the dividend and determine if the divisor was smaller in value. In this example the divisor is larger, so the quotient is zero. Next, you examine the two most significant digits. Again the divisor is larger, so the quotient is again zero. Finally, you examine the whole dividend and discover it is approximately four times the divisor in value. Therefore, you give the quotient a value of 4. Next, you subtract the product of 45 and 4 (180) from the dividend. The difference of one represents a fraction of the divisor. This fraction is added to the quotient to produce the correct answer of $4\frac{1}{45}$.

Binary division is performed in a similar manner. However, binary division is a simpler process since the number base is two rather than ten. First, let's divide 100011_2 by 101_2 .

		000111	Quotient
Divisor:	101	$\overline{)100011}$	Dividend
		101	
		$\underline{111}$	Remainder
		101	
		$\underline{101}$	Remainder
		101	
		$\underline{0}$	Remainder

Using long division, you examine the dividend beginning with the MSB and determine the number of bits required to exceed the value of the divisor. When you find this value, place a one in the quotient and subtract the divisor from the selected dividend value. Then carry the next least significant bit in the dividend down to the remainder. If you can subtract the divisor from the new remainder, place a one in the quotient. Then subtract the divisor from the remainder and carry the next least signifi-

cant bit in the dividend (LSB in this example) down to the remainder. If the divisor can be subtracted from the new remainder, place a one in the quotient and subtract the divisor from the remainder. Continue the process until all of the dividend bits have been carried down. Then express any remainder as a fraction of the divisor in the quotient. Thus, 100011_2 divided by 101_2 equals 111_2 . You can verify the answer by converting the binary numbers to decimal.

To make sure you fully understand binary division, work out the following examples of long division. Divide 101000_2 by 1000_2 and then divide 100111_2 by 110_2 .

		000101	Quotient
Divisor	1000	$\overline{)101000}$	Dividend
		1000 ↓↓	
		1000	Remainder
		1000	
		0	Remainder

		000110.1	Quotient
Divisor:	110	$\overline{)100111.0}$	Dividend
		110 ↓ ↓	
		111	Remainder
		110 ↓ ↓	
		110	Remainder
		110	
		0	Remainder

In the second example, the quotient was not a whole number, but rather a whole number plus a fraction (remainder divided by the divisor). The answer $110-11/110$ is correct. You could have left the answer in this form or, as in the example, continue the division process until the remainder was zero. This is made possible by adding a sufficient number of zeros after the binary point to permit division by the divisor. In the previous example, only one zero was added after the binary point. As you learned in Unit 1, adding zeros after the binary point will not affect the value of the number. Note that some numbers cannot be solved in this manner (e.g., decimal $1/3$).

$$\begin{array}{r} 101 \\ 1000 \overline{)101000} \\ \underline{1000} \\ 1000 \\ \underline{1000} \\ 0 \end{array}$$

$$\begin{array}{r} 110.1 \\ 110 \overline{)100111.0} \\ \underline{110} \\ 111 \\ \underline{110} \\ 110 \\ \underline{110} \\ 0 \end{array}$$

Representing Negative Numbers

Until now, we have been examining binary arithmetic using unsigned numbers. However, when you perform some arithmetic operations with a microprocessor, you must be able to express both positive and negative (signed) numbers. Over the years three methods have been developed for representing signed numbers. Of these, only one method has survived. The two older methods will be examined first, and then the system that is used today.

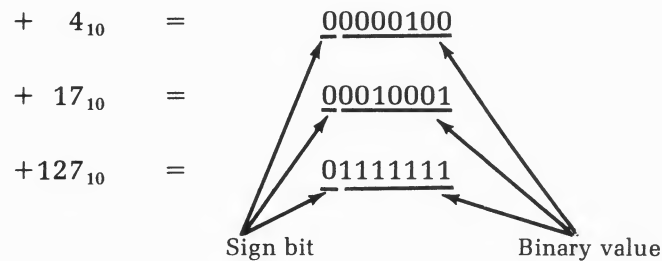
SIGN AND MAGNITUDE. Using this system, a binary number contained both the sign (+ or -) and the value of the number. Therefore, positive and negative values were expressed as follows:

$$\begin{array}{rcl} +45_{10} & = & \underline{00101101}_2 \\ & \nearrow \quad \nwarrow & \\ & \text{SIGN} \quad \text{MAGNITUDE} & \\ & \nwarrow \quad \nearrow & \\ -45_{10} & = & \underline{10101101}_2 \end{array}$$

The MSB of the binary number indicated the sign, while the remaining bits contained the value of the number. As you can see, a zero sign bit indicated a positive value, while a one sign bit indicated a negative value.

While this method of representing negative numbers may seem logical, its popularity was short-lived. Because it required complex and slow arithmetic circuitry, it was abandoned long before microprocessors were invented.

ONE'S COMPLEMENT. Another method of representing negative numbers became popular in the early days of computers. It was called the one's complement method. Using this system, positive numbers were represented in the same way as in the sign-magnitude system. That is, the MSB in any number was considered to be a sign bit. A sign bit of 0 represented positive. Using 8-bit numbers, positive values were represented like this:



Negative numbers were represented by the **one's complement** of the positive value. The one's complement of a number is formed by changing all 0's to 1's and all 1's to 0's. As shown above, +4₁₀ is represented as 0 0000100₂. By changing all 0's to 1's and all 1's to 0's, the representation for -4₁₀ was formed. In this case:

$$- 4_{10} = \underline{1} \underline{1111011}_2$$

Notice that all the bits, including the sign bit, were inverted. In the same way:

$$- 17_{10} = \underline{1} \underline{1101110}_2$$

$$-127_{10} = \underline{1} \underline{0000000}_2$$

The one's complement method is not used for representing signed numbers in microprocessors. However, as you will see later, you may still be called upon to find the one's complement of a number. Remember, you do this by simply changing all 0's to 1's and all 1's to 0's.

Figure 3-4 shows an interesting relationship. In the first column, 8-bit patterns of 0's and 1's are shown. The second column shows the decimal number that each pattern represents if you consider the pattern to be an unsigned binary number. Notice that an 8-bit pattern can represent unsigned numbers between 0 and 255_{10} .

The third column shows the decimal number that each pattern represents if you consider the pattern to be a one's complement binary number. Notice that the range of numbers is from -127_{10} to $+127_{10}$. Notice also that there are two representations of zero. The pattern $0000\ 0000_2$ represents $+0$ while its one's complement ($1111\ 1111_2$) represents -0 .

TWO'S COMPLEMENT. The method used to represent signed numbers in microprocessors is called two's complement. In this system, positive numbers are represented just as they were with the sign-and-magnitude method and the one's complement method. That is, it uses the same bit pattern for all positive values up to $+127_{10}$. However, negative numbers are represented as the two's complement of positive numbers.

The two's complement of a number is formed by taking the one's complement and then adding 1. For example if you work with 8-bit numbers and use the two's complement system, $+4_{10}$ is represented by 00000100_2 . To find -4_{10} , you must take the two's complement of this number. You do this by first taking the one's complement, which is 11111011_2 . Next, add 1 to form the two's complement:

$$\begin{array}{r} 11111011_2 \\ + \quad \quad 1 \\ \hline 11111100_2 \end{array}$$

Thus, the two's complement representation of -4_{10} is 11111100_2 .

To be sure you have the idea, look at a second example: how do you express -17_{10} as an 8-bit two's complement number? Start with the two's complement representation of $+17_{10}$, which is 00010001_2 . Take the one's complement by changing all 0's to 1's and 1's to 0's. Thus, the one's complement of $+17_{10}$ is 11101110_2 . Next, find the two's complement by adding 1:

$$\begin{array}{r} 11101110_2 \\ + \quad \quad 1 \\ \hline 11101111_2 \end{array}$$

BIT PATTERN	UNSIGNED BINARY	1's COMPLEMENT
00000000	0	+0
00000001	1	+1
00000010	2	+2
00000011	3	+3
.	.	.
.	.	.
.	.	.
.	.	.
01111100	124	+124
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-127
10000001	129	-126
10000010	130	-125
10000011	131	-124
.	.	.
.	.	.
.	.	.
.	.	.
11111100	252	-3
11111101	253	-2
11111110	254	-1
11111111	255	-0

Figure 3-4

Table of bit pattern values for unsigned binary numbers and 1's complement numbers.

Figure 3-5 compares unsigned, two's complement, and one's complement numbers. Several 8-bit patterns are shown on the left. The other three columns show the decimal number represented by these patterns.

Notice that the range of 8-bit two's complement numbers is from -128_{10} to $+127_{10}$. Notice also that there is only one representation for 0.

If this table included all 256_{10} possible 8-bit patterns, you could look up any pattern to see what number it represents. The patterns which have 0 as their MSB are easy to determine without a table. The pattern represents the binary number directly. But what decimal number is represented by the two's complement number 11110011? You should know that this represents some negative number because the MSB is a 1.

Actually, you can determine the value very easily by simply taking the two's complement to find the equivalent positive number. Remember, you find the two's complement, by taking the one's complement and adding 1. The one's complement is 00001100₂. Thus, the two's complement is:

$$\begin{array}{r} 00001100_2 \\ + \quad \quad 1 \\ \hline 00001101_2 \end{array} \quad \text{or } +13_{10}$$

Since the two's complement of 11110011₂ represents $+13_{10}$, then 11110011₂ must equal -13_{10} .

BIT PATTERN	UNSIGNED BINARY	2's COMPLEMENT	1's COMPLEMENT
00000000	0	0	+0
00000001	1	+1	+1
00000010	2	+2	+2
00000011	3	+3	+3
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
01111100	124	+124	+124
01111101	125	+125	+125
01111110	126	+126	+126
01111111	127	+127	+127
10000000	128	-128	-127
10000001	129	-127	-126
10000010	130	-126	-125
10000011	131	-125	-124
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
11111100	252	-4	-3
11111101	253	-3	-2
11111110	254	-2	-1
11111111	255	-1	-0

Figure 3-5

Table of bit pattern values for unsigned binary, 2's complement and 1's complement numbers.

Self-Test Review

1. _____ and _____ are the two basic operations that can be performed with a number system.

2. Add the following binary numbers.

A.
$$\begin{array}{r} 10011011 \\ +00010111 \\ \hline \end{array}$$

B.
$$\begin{array}{r} 11000110 \\ +00110001 \\ \hline \end{array}$$

C.
$$\begin{array}{r} 10000110 \\ +00110110 \\ \hline \end{array}$$

3. Subtract the following binary numbers.

A.
$$\begin{array}{r} 11011011 \\ -10110010 \\ \hline \end{array}$$

B.
$$\begin{array}{r} 10001011 \\ -10000001 \\ \hline \end{array}$$

C.
$$\begin{array}{r} 11011001 \\ -00111011 \\ \hline \end{array}$$

4. Multiply the following binary numbers.

A.
$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline \end{array}$$

B.
$$\begin{array}{r} 1101 \\ \times 1001 \\ \hline \end{array}$$

C.
$$\begin{array}{r} 1100 \\ \times 1100 \\ \hline \end{array}$$

5. Solve for the quotient in the following groups.

A.
$$\begin{array}{r} 101 \overline{)1001011} \end{array}$$

B.
$$\begin{array}{r} 11 \overline{)111001} \end{array}$$

C.
$$\begin{array}{r} 1101 \overline{)11110111} \end{array}$$

6. 10001111_2 represents decimal _____ in sign/magnitude notation.

7. The 1's complement of 00010110_2 is _____.
8. The 2's complement of 00010110_2 is _____.
9. The 2's complement number 11100110 represents the decimal number _____.
10. Find the signed decimal equivalents of the following two's complement numbers.

<u>Two's Complement Number</u>	<u>Decimal Number</u>
00000111	
10000111	
11111111	
01110000	
10000000	

11. Find the two's complement representation for the following signed decimal numbers.

<u>Decimal Number</u>	<u>Two's Complement Number</u>
+32	
-32	
+73	
- 7	
-120	

Answers

- | | | | |
|----|------------------------|--------------------------|--------------------------|
| 1. | Addition, subtraction. | | |
| 2. | A. | Carry: | 00011111 |
| | | Addend: | 10011011 |
| | | Augend: | + 00010111 |
| | | Sum: | <u>10110010</u> |
| | B. | 11110111. | |
| | C. | 10111100. | |
| 3. | A. | Minuend after borrow: | 1 0 10 1 1 0 1 1 |
| | | Minuend: | 1 1 0 1 1 0 1 1 |
| | | Subtrahend: | <u>- 1 0 1 1 0 0 1 0</u> |
| | | Difference: | 1 0 1 0 0 1 |
| | B. | 1010. | |
| | C. | 10011110. | |
| 4. | A. | Multiplicand: | 1011 |
| | | Multiplier: | <u>× 1101</u> |
| | | First partial product: | 1011 |
| | | Second partial product: | <u>0000</u> |
| | | Carry: | <u>0000</u> |
| | | Sum of partial products: | 01011 |
| | | Third partial product: | <u>101100</u> |
| | | Carry: | <u>01000</u> |
| | | Sum of partial products: | 110111 |
| | | Fourth partial product: | <u>1011000</u> |
| | | Carry: | <u>1110000</u> |
| | | Final product: | 10001111 |
| | B. | 1110101. | |
| | C. | 10010000 | |

5.	A.	Divisor:	101	$\overline{)0001111}$	Quotient
				1001011	Dividend
				<u>101</u>	
				1000	Remainder
				<u>101</u>	
				111	Remainder
				<u>101</u>	
				101	Remainder
				<u>101</u>	
				0	Remainder

B. 10011.

C. 10011.

6. -15.

7. 11101001₂.

8. 11101010₂.

9. First, find the two's complement of 11100110 by changing 1's to 0's; 0's to 1's; and adding 1:

$$\begin{array}{r} 00011001 \\ \underline{1} \\ 00011010 \end{array}$$

Since this number represents +26₁₀, the original number must have represented -26₁₀.

10.	<u>Two's Complement Number</u>	<u>Decimal Number</u>
	00000111	+7
	10000111	-121
	11111111	-1
	01110000	+112
	10000000	-128

11.	<u>Decimal Number</u>	<u>Two's Complement Number</u>
	+32	00100000
	-32	11100000
	+73	01001001
	-7	11111001
	-120	10001000

TWO'S COMPLEMENT ARITHMETIC

In the previous section, you saw that signed numbers are represented in microprocessors in two's complement form. In this section you will see why.

In digital electronic devices such as computers, simple circuits cost less and operate faster than more complex ones. Two's complement numbers are used with arithmetic because they allow the simplest, cheapest, and fastest circuits.

A characteristic of the two's complement system is that both signed and unsigned numbers can be added by the same circuit. For example, suppose you wish to add the **unsigned** numbers 132_{10} and 14_{10} . The addition looks like this:

Addend:	10000100_2	132_{10}
Augend:	00001110_2	$+ 14_{10}$
Sum:	10010010_2	146_{10}

As you saw in the previous unit, the microprocessor has an ALU circuit that can add unsigned binary numbers in this way. The adder in the ALU is designed so that when the bit pattern 10000100 appears at one input and 00001110 appears at the other, the bit pattern 10010010 appears at the output.

The question arises, "How does the ALU know that the bit patterns at the inputs represent unsigned numbers and not two's complement numbers?" The answer is "it doesn't." The ALU always adds as if the inputs were unsigned binary numbers. Nevertheless, it still produces the correct sum even if the inputs are signed two's complement numbers.

Look at the example given above. If you assume that the inputs are two's complement signed numbers, then the addend, augend, and sum are:

Addend:	10000100_2	-124_{10}
Augend:	00001110_2	$+ 14_{10}$
Sum:	10010010_2	-110_{10}

Notice that the bit patterns are the same. Only the meaning of the bit patterns has changed. In the first example, we assumed that the bit patterns represented unsigned numbers and the adder produced the proper unsigned result. In the second example, we assumed that the bit patterns represented signed numbers. Again, the adder produced the proper signed result.

This proves a very important point. The adder in the ALU always adds bit patterns as if they are unsigned binary numbers. It is our interpretation of these bit patterns that decides if unsigned or signed numbers are indicated. The beauty of two's complement is that the bit patterns can be interpreted either way. This allows us to work with either signed or unsigned numbers without requiring different circuits for each.

Two's complement arithmetic also simplifies the arithmetic logic unit in another way. All microprocessors have a subtract instruction. Thus, the ALU must be able to subtract one number from another. However, if this required a separate subtraction circuit, the complexity and cost of the ALU would be increased. Fortunately, two's complement arithmetic allows the ALU to perform a subtract operation using an adder circuit. That is, the MPU uses the same circuit for both addition and subtraction.

The MPU performs subtraction by a binary addition process. To see why this works, it may be helpful to look at a similar process with the decimal number system. The decimal equivalent of two's complement is called ten's complement. Since you are more familiar with the decimal number system, briefly examine ten's complement arithmetic.

Ten's Complement Arithmetic

An easy way to illustrate ten's complement is to consider an analogy. Visualize the odometer or mileage indicator on your car. Generally, this is a six-digit device that indicates mileage between 00,000.0 and 99,999.9 miles. Let's ignore the tenths digit and concentrate on the other five.


In an automobile, the register generally operates in only one direction (forward). However, consider what happens if it is turned backwards instead. Starting at +3 miles, the count proceeds backwards as follows:

00,003
00,002
00,001
00,000
99,999
99,998
99,997
etc.

It is easy to visualize that 99,999 represents -1 mile. Also, 99,998 represents -2 miles; 99,997 represents -3 miles; etc. This is how signed numbers are represented in ten's complement form.

Once you accept this system for representing positive and negative numbers, you can perform arithmetic with these signed numbers. For example, if you add +3 and -2, the result should be +1. Using the system developed above, +3 is represented by 00003 while -2 is represented by 99,998. Thus, the addition looks like this:

$$\begin{array}{r}
 00003 \quad +3 \\
 +99998 \quad -2 \\
 \hline
 100001 \quad +1
 \end{array}$$

 Discard final carry.

If you now discard the final carry on the left in the sum, the answer is 00001, the representation of +1. You can also find the ten's complement of a digit by subtracting the digit from ten. For example, the ten's complement of 6 is 4 since $10 - 6 = 4$. To complement a number containing more than one digit, raise ten to a power equal to the total number of digits, then subtract the number from it. For example, to obtain the ten's complement of 654_{10} , first raise ten to the third power since there are three digits in the number. Then, subtract 654 from the result.

$$\begin{array}{r}
 10^3 = 1000 \\
 -654 \\
 \hline
 346
 \end{array}$$

Thus, the ten's complement of 654_{10} is 346_{10} .

Once you find the ten's complement, you can subtract one number from another by an indirect method using only addition. Since childhood you have subtracted like this:

$$\begin{array}{r}
 \text{Minuend:} \quad 973 \\
 \text{Subtrahend:} \quad -654 \\
 \hline
 \text{Difference:} \quad 319
 \end{array}$$

However, you can arrive at the same answer by using the ten's complement of the subtrahend and adding. Recall that the ten's complement of 654_{10} is 346_{10} . Let's compare these two methods of subtraction:

<u>STANDARD METHOD</u>		<u>TEN'S COMPLEMENT METHOD</u>	
Minuend	973	973	Minuend
Subtrahend	<u>-654</u>	<u>+346</u>	Ten's complement of subtrahend
Difference	319	1319	Difference
		↑	
		Discard final carry	

Notice that when you use the ten's complement method, the answer is too large by 1000_{10} . However, you can still arrive at the correct answer by simply discarding the final carry.

While the ten's complement method of subtraction works, it is not used because it is more complex than the standard method. In fact, it does not eliminate subtraction entirely since the ten's complement itself is found by subtraction.

The binary equivalent of ten's complement is two's complement. It overcomes the disadvantage of ten's complement in that the two's complement can be formed without any subtraction at all. Recall that you can form the two's complement of a binary number by changing all 0's to 1's, all 1's to 0's and then adding 1. Let's examine two's complement arithmetic in more detail.

Two's Complement Subtraction


As in ten's complement arithmetic, you can form the two's complement by subtracting from a power of the base (two). However, because the MPU cannot subtract directly, it uses the method given earlier for finding the two's complement. Once the two's complement is formed, the MPU can perform subtraction indirectly by adding the two's complement of the subtrahend to the minuend.

To illustrate this point, look at the following two ways of subtracting 26_{10} from 69_{10} . The two numbers are expressed as they would appear to an 8-bit microprocessor. The standard method of subtraction looks like this:

Minuend:	01000101 ₂	69
Subtrahend:	<u>-00011010₂</u>	<u>-26</u>
Difference:	00101011 ₂	43

While this method works fine on paper, it's of little use to the microprocessor since the MPU has no subtract circuitry. However, the MPU can still perform subtraction by the indirect method of adding the two's complement of the subtrahend to the minuend:

	Minuend:	01000101
Two's complement of	Subtrahend:	<u>+11100110</u>
	Difference:	100101011

 Discard final carry

This illustrates a major reason for using the two's complement system to represent signed numbers. It allows the MPU to perform subtraction and addition with the same circuit.

The method that the MPU uses to perform subtraction is of little importance to the user of microprocessors. Most microprocessors have a subtract instruction. This instruction is used like any other without regard for how the operation is implemented internally. When the subtract instruction is implemented, the MPU automatically takes care of operations like complementing the subtrahend, adding, and discarding the carry. The procedure has been explained here so you can appreciate the importance of two's complement arithmetic.

Arithmetic With Signed Numbers

There are many applications in which the microprocessor must work with signed numbers. In these cases, signed numbers are represented in two's complement form. While this greatly simplifies the circuitry of the MPU, it places an extra burden on the user. The programmer must ensure that all signed numbers are entered into the microprocessor in two's complement form. Also, the resulting data produced by the MPU may be in two's complement form. Here's how an 8-bit MPU handles signed numbers.

Adding Positive Numbers. Assume that the MPU is to add the two positive numbers +7 and +3. Since an 8-bit MPU is assumed, the arithmetic operation looks like this:

<u>00000111</u>	+ 7
<u>+00000011</u>	+ 3
<u>00001010</u>	+ 10

The sign bits are underlined. Remember, when representing signed numbers, that the MSB is the sign bit. A 0 represents “+” and a 1 represents “-.” In this example, you added +7 and +3 to form a sum of +10₁₀. You know that all three numbers are positive since the MSB’s are all 0’s.

While this operation seems straightforward enough, it is easy for the unwary to make an error when adding positive numbers. Remember, the highest 8-bit positive number you can represent in two’s complement form is +127₁₀. If the sum exceeds this value, an error occurs. For example, suppose you attempt to add +65₁₀ to +67₁₀. The MPU adds the numbers as if they are unsigned binary:

$$\begin{array}{r} \underline{0}1000001 \\ \underline{0}1000011 \\ \hline 10000100 \end{array}$$

If the answer is interpreted as a two’s complement number, an error has occurred. You have added two positive numbers and yet the answer appears to be negative since the MSB of the sum is 1. This is called a two’s complement overflow. It occurs when the sum exceeds +127₁₀. Many microprocessors have a way of detecting this condition. We will discuss this in more detail in a future unit.

Adding Positive and Negative Numbers. The real beauty of the two’s complement system is illustrated when you add numbers with unlike signs. For example, assume that an 8-bit microprocessor is to add +7 and -3. Remember, since these are signed numbers, they must be represented in two’s complement form. That is, +7 is represented as 00000111₂ while -3 is represented as 1111101₂. If these two numbers are added, the sum will be:

$$\begin{array}{rcl} \text{Addend:} & 00000111 & (+7) \\ \text{Augend:} & +1111101 & +(-3) \\ \text{Sum:} & \underline{100000100} & (+4) \end{array}$$

↑
Discard final carry

Notice that the sum is correct if you ignore the final carry bit. Keep in mind that the MPU adds the two numbers as if they were unsigned binary numbers. It is merely our interpretation of the answer that makes the system work for signed numbers.

The system also works when the negative number is larger. For example, when -9 is added to $+8$ the result should be -1 . Remember, the signed numbers must be represented in two's complement form:

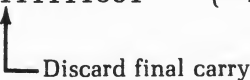
Addend:	11110111	(-9)
Augend:	<u>00001000</u>	<u>+(+8)</u>
Sum:	11111111	-1

Notice that the sum is the two's complement representation for -1 .

Adding Negative Numbers. The final case involves two negative numbers. If both numbers are negative, then the sum should also be negative.

For example, suppose the MPU is to add -3 to -4 . Obviously, the result should be -7 . The two signed numbers must be represented in two's complement form. That is, -3 must be represented as 11111101_2 while -4 must be represented as 11111100_2 . The MPU adds these two bit patterns as if they were unsigned binary numbers. Thus the result is:

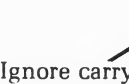

Addend:	11111101	(-3)
Augend:	<u>+11111100</u>	<u>+(-4)</u>
Sum:	111111001	(-7)


 Discard final carry

Once again, the answer is correct if you ignore the final carry bit.

When you add two negative numbers, you must remember the capacity of the MPU. The largest negative number that can be represented by eight bits is -128_{10} . If the sum exceeds this value, the sum will appear to be in error. For example, suppose you add -120_{10} to -18_{10} .

	10001000	(-120)
	<u>11101110</u>	<u>+(-18)</u>
	101110110	


 Ignore carry
 
 Sign bit

Notice that the sign bit in the sum is 0, representing a positive number. Thus, the MPU has added two negative numbers and has produced a positive result. This apparent error is caused by exceeding the 8-bit capacity. This is another example of two's complement overflow.

Self-Test Review

12. In microprocessors, signed numbers are represented in _____ form.
13. The ALU adds bit patterns as if they represent _____ binary numbers.
14. When a microprocessor executes a subtract instruction, what operations are actually performed inside the MPU?
15. What is the largest 8-bit positive number that can be represented in two's complement form?
16. When you are adding two positive numbers, what is meant by two's complement overflow?
17. If $+19_{10}$ and -21_{10} are added by an 8-bit microprocessor, the two's complement result will be _____.
18. Can two's complement overflow occur when two negative numbers are added?
19. A microprocessor adds 10001110_2 to 00010001_2 . If these are unsigned binary numbers, the resulting bit pattern will be _____. If these are two's complement numbers, the resulting bit pattern will be _____.
20. If the bit patterns in question 19 represent unsigned numbers, the resulting bit pattern represents decimal _____.
21. If the bit patterns in question 19 represent two's complement numbers, the resulting bit pattern represents decimal _____.

Answers

12. Two's complement.
13. Unsigned.
14. The following operations occur:
 1. The MPU complements the subtrahend by changing 0's to 1's and 1's to 0's.
 2. One is added to the complemented subtrahend to form the two's complement.
 3. The two's complement of the subtrahend is added to the minuend.
15. 01111111_2 or $+127_{10}$.
16. When you add positive numbers, two's complement overflow occurs when the sum exceeds $+127_{10}$.
17. 11111110_2 or -2_{10} .
18. Yes. When you add negative numbers, two's complement overflow occurs when the sum exceeds -128_{10} .
19. In either case, the resulting bit pattern will be 10011111.
20. 159_{10} .
21. -97_{10} .

BOOLEAN OPERATIONS

Along with the basic mathematical processes examined earlier, the microprocessor can manipulate binary numbers logically. This system was conceived using the theorems developed by the mathematician George Boole. As a result, this branch of binary mathematics is given the name Boolean Algebra. In this section, the Boolean operations performed by the microprocessor will be examined. A more detailed description of Boolean Algebra is provided in the Heathkit Continuing Education Series course titled "Digital Techniques."

AND Operation

The AND function produces the logical product of two or more logic variables. That is, the logical product of an AND operation is logic 1 if all of the variable inputs are logic 1. If any of the input variables are logic 0, the logical product is 0. This process can be represented by the formula $A \cdot B = C$, where A and B represent input variables (logic 1 or 0) and C represents the output or logical product of the AND operation. The AND function is designated by a dot between the variables. Do not confuse it with the mathematical multiplication sign.

Figure 3-6 is a "truth" table for a two-variable AND function. The 1's and 0's represent all of the possible logic combinations. Thus, you can see that the AND function is a sort of "all or nothing" operation. Unless all the input variables are logic 1, the output cannot be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3-6
Truth Table for an AND function.

When the microprocessor implements the logic AND operation, one 8-bit binary number is ANDed with a second 8-bit binary number. Refer to Figure 3-7 for an illustration of this process.

<u>8-BIT NUMBER</u>			<u>8-BIT NUMBER</u>		<u>RESULT OF AND OPERATION</u>
MSB	1	•	1	=	1 MSB
	0	•	0	=	0
	0	•	1	=	0
	1	•	0	=	0
	1	•	1	=	1
	0	•	1	=	0
	1	•	0	=	0
LSB	0	•	0	=	0 LSB

Figure 3-7
8-bit logic AND operation.

Although more than two logic variables can be ANDed together, the microprocessor operates on only two variables at a time. Now try one more example of the AND operation. AND 10011101 with 11000110.

$$\begin{array}{rcl}
 1 \cdot 1 & = & 1 \quad \text{MSB} \\
 0 \cdot 1 & = & 0 \\
 0 \cdot 0 & = & 0 \\
 1 \cdot 0 & = & 0 \\
 1 \cdot 0 & = & 0 \\
 1 \cdot 1 & = & 1 \\
 0 \cdot 1 & = & 0 \\
 1 \cdot 0 & = & 0 \quad \text{LSB}
 \end{array}$$

OR Operation

The OR (sometimes known as inclusive OR) function produces the logical sum of two or more logic variables. That is, the logical sum of an OR operation is logic 1 if either input is logic 1. The logical sum is 0 if **all** of the input variables are logic 0. This process can be represented by the formula $A + B = C$, where A and B represent input variables and C represents the output or logical sum of the OR operation. The OR function is designated by a plus sign (or a circled dot \odot) between the variables. Do not confuse the plus sign with the mathematical add sign.

Figure 3-8 is a “truth” table for a two-variable OR function. The 1’s and 0’s represent all of the possible logic combinations. Thus, you can see that the OR function is a sort of “either or both” operation. If either or both input variables are logic 1, the output must be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

Figure 3-8
Truth Table for an OR function.

When the microprocessor implements the logic OR operation, one 8-bit binary number is ORed with a second 8-bit binary number. Refer to Figure 3-9 for an illustration of this process.

<u>8-BIT NUMBER</u>			<u>8-BIT NUMBER</u>		<u>RESULT OF OR OPERATION</u>	
MSB	1	+	1	=	1	MSB
	0	+	0	=	0	
	0	+	1	=	1	
	1	+	0	=	1	
	1	+	1	=	1	
	0	+	1	=	1	
	1	+	0	=	1	
LSB	0	+	0	=	0	LSB

Figure 3-9
8-bit logic OR operation.

As with the AND function, two or more logic variables can be ORed together. However, the microprocessor operates on only two variables at a time. Now try one more example of the OR operation. OR 10011101 with 11000101.

$$\begin{array}{rcl}
 1 + 1 & = & 1 \quad \text{MSB} \\
 0 + 1 & = & 1 \\
 0 + 0 & = & 0 \\
 1 + 0 & = & 1 \\
 1 + 0 & = & 1 \\
 1 + 1 & = & 1 \\
 0 + 0 & = & 0 \\
 1 + 1 & = & 1 \quad \text{LSB}
 \end{array}$$

Exclusive OR Operation

The Exclusive OR (EOR or XOR) function performs a logical test for “equalness” between two logic variables. That is, if two variable inputs are equal (both logic 1 or 0), the output or result of the EOR operation is logic 0. If the inputs are not equal (one is logic 1, the other logic 0) the output is logic 1. This can be represented by the formula $A \oplus B = C$, where A and B represent input variables and C represents the output or result. The EOR function is designated by a circled plus sign between the variables.

Figure 3-10 is a “truth” table for the EOR function. The 1’s and 0’s represent all of the possible logic combinations. You can see that the EOR function is a sort of “either but not both” operation. That is, either input can be logic 1 or 0, but not both for a logic 1 output.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3-10

Truth Table for an EOR function.

When the microprocessor implements the logic EOR operation, one 8-bit binary number is exclusively ORed with a second 8-bit number. Refer to Figure 3-11 for an illustration of this process.

	<u>8-BIT</u> <u>NUMBER</u>		<u>8-BIT</u> <u>NUMBER</u>		<u>RESULT OF</u> <u>EOR OPERATION</u>	
MSB	1	\oplus	1	=	0	MSB
	0	\oplus	0	=	0	
	0	\oplus	1	=	1	
	1	\oplus	0	=	1	
	1	\oplus	1	=	0	
	0	\oplus	1	=	1	
	1	\oplus	0	=	1	
LSB	0	\oplus	0	=	0	LSB

Figure 3-11
8-bit logic EOR operation.

Now try one more example of the EOR operation. EOR 10011101_2 with 11000101_2 .

$$\begin{array}{rcl}
 1 \oplus 1 & = & 0 \quad \text{MSB} \\
 0 \oplus 1 & = & 1 \\
 0 \oplus 0 & = & 0 \\
 1 \oplus 0 & = & 1 \\
 1 \oplus 0 & = & 1 \\
 1 \oplus 1 & = & 0 \\
 0 \oplus 0 & = & 0 \\
 1 \oplus 1 & = & 0 \quad \text{LSB}
 \end{array}$$

Invert Operation

The invert operation performs a direct complement of a single input variable. That is, a logic 1 input will produce a logic 0 output. This process can be represented by the truth table in Figure 3-12.

INPUT	OUTPUT
A	\overline{A}
1	0
0	1

Figure 3-12

Truth Table for an invert function.

Note that the complement of A is \overline{A} . The bar above the A indicates that A has been inverted, and is read “not A .” Therefore, the complement of A is “not A ” (\overline{A}).

When the microprocessor implements the logic invert operation, the 8-bit binary number is complemented. This operation is also known as 1’s complement. Thus, the complement of 11010110_2 is 00101001_2 . As with the previous logic operations, the invert function operates on each individual bit of the 8-bit number.

Self-Test Review

22. The result of an AND operation is binary 1 when:
- A. All inputs are binary 0.
 - B. Any one input is binary 0.
 - C. All inputs are binary 1.
 - D. Any one input is binary 1.
23. Perform the AND operation on the following 8-bit number pairs.
- A. 11010110 and 10000111.
 - B. 00110011 and 11110000.
 - C. 10101010 and 11011011.
24. The result of an OR operation is binary 0 when:
- A. All inputs are binary 1.
 - B. All inputs are binary 0.
 - C. Any one input is binary 1.
 - D. Any one input is binary 0.
25. Perform the OR operation on the following 8-bit number pairs.
- A. 11010110 and 10000111.
 - B. 00110011 and 11110000.
 - C. 10101010 and 11011011.
26. The result of an XOR operation is binary 0 if the inputs are:
- A. Equal.
 - B. Not equal.

27. The symbol for the EOR operation is:
- A. \cdot
 - B. $+$
 - C. \oplus
 - D. \times
28. Perform the EOR operation on the following 8-bit number pairs.
- A. 11010110 and 10000111.
 - B. 00110011 and 11110000.
 - C. 10101010 and 11011011.
29. \overline{A} represents the _____ of A.
- A. Sum.
 - B. Product.
 - C. Complement.
 - D. Supplement.
30. Perform the invert operation on the following 8-bit numbers.
- A. 11010110.
 - B. 00110011.
 - C. 10101010.

Answers

22. C. All inputs are binary 1.

23. A. $1 \cdot 1 = 1$
 $1 \cdot 0 = 0$
 $0 \cdot 0 = 0$
 $1 \cdot 0 = 0$
 $0 \cdot 0 = 0$
 $1 \cdot 1 = 1$
 $1 \cdot 1 = 1$
 $0 \cdot 1 = 0$

B. 00110000.

C. 10001010.

24. B. All inputs are binary 0.

25. A. $1 + 1 = 1$
 $1 + 0 = 1$
 $0 + 0 = 0$
 $1 + 0 = 1$
 $0 + 0 = 0$
 $1 + 1 = 1$
 $1 + 1 = 1$
 $0 + 1 = 1$

B. 11110011.

C. 11111011.

26. A. Equal.

27. C. \oplus
28. A. $1 \oplus 1 = 0$
 $1 \oplus 0 = 1$
 $0 \oplus 0 = 0$
 $1 \oplus 0 = 1$
 $0 \oplus 0 = 0$
 $1 \oplus 1 = 0$
 $1 \oplus 1 = 0$
 $0 \oplus 1 = 1$
- B. 11000011.
- C. 01110001.
29. C. Complement.
30. A. 00101001.
- B. 11001100.
- C. 01010101.

EXPERIMENT 4

Perform Experiment 4 in Unit 7 of this course. After you finish the experiment, return to this Unit and complete the Unit Examination.

UNIT EXAMINATION

1. Add 10010110_2 to 1101_2 .
2. Subtract 1011_2 from 10110110_2 .
3. Multiply 1001_2 by 1100_2 .
4. Divide 100111_2 with 110_2 .
5. The 1's complement of 00110110_2 is _____.
6. The 2's complement of 00110110_2 is _____.
7. Using 2's complement arithmetic, add $+75_{10}$ to -6_{10} .
8. Using 2's complement arithmetic, add -35_{10} to -75_{10} .
9. Using 2's complement arithmetic, subtract -15_{10} from -85_{10} .
10. The truth table Figure 3-13 represents the logical _____ function.

INPUT		OUTPUT
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

Figure 3-13
Truth Table for Exam Question 10.

11. Logically AND 11011010 with 10010110 .
12. Logically OR 11011010 with 10010110 .
13. Logically EOR 11011010 with 10010110 .
14. Logically invert 11011010 .

EXAMINATION ANSWERS

1. Carry: 00011100
 Addend: 10010110
 Augend: + 1101
 Sum: 10100011

2. Minuend after borrow: 1 0 1 0 10 0 10 10
 Minuend: 1 0 1 1 0 1 1 0
 Subtrahend: 1 0 1 1
 Difference: 1 0 1 0 1 0 1 1

3. Multiplicand: 1001
 Multiplier: $\times 1100$
 First partial product: 0000
 Second partial product: 0000
 Carry: 0000
 Sum of partial products: 00000
 Third partial product: 1001
 Carry: 00000
 Sum of partial products: 100100
 Fourth partial product: 1001
 Carry: 000000
 Final product: 1101100

4. Divisor: 110 $\overline{)000110.1}$ Quotient
 100111.0 Dividend
 110
 111 Remainder
 110
 110 Remainder
 110
 0 Remainder

5. 11001001₂.

6. 11001010₂.

$$7. \quad +75_{10} = 01001011_2.$$

$$+ 6_{10} = 00000110_2.$$

$$- 6_{10} = 11111010_2.$$

11111010	Carry
01001011	Addend
+ 11111010	Augend
<u>101000101</u>	Sum
01000101	Corrected sum

$$01000101_2 = +69_{10}.$$

$$8. \quad +35 = 00100011_2.$$

$$-35 = 11011101_2.$$

$$+75_{10} = 01001011_2.$$

$$-75_{10} = 10110101_2.$$

11111101	Carry
11011101	Addend
+ 10110101	Augend
<u>110010010</u>	Sum
10010010	Correct sum

$$10010010_2 = -110_{10}.$$

$$9. \quad +85_{10} = 01010101_2.$$

$$-85_{10} = 10101011_2.$$

$$+15_{10} = 00001111_2.$$

$$-15_{10} = 11110001_2.$$

10101011	}	00001111	Carry
-11110001		10101011	Addend
<u>10111010</u>		+00001111	Augend
		010111010	Sum
		10111010	Corrected sum

$$10111010_2 = -70_{10}.$$

10. Exclusive OR (XOR or EOR).

11. $1 \cdot 1 = 1$
 $1 \cdot 0 = 0$
 $0 \cdot 0 = 0$
 $1 \cdot 1 = 1$
 $1 \cdot 0 = 0$
 $0 \cdot 1 = 0$
 $1 \cdot 1 = 1$
 $0 \cdot 0 = 0$

12. $1 + 1 = 1$
 $1 + 0 = 1$
 $0 + 0 = 0$
 $1 + 1 = 1$
 $1 + 0 = 1$
 $0 + 1 = 1$
 $1 + 1 = 1$
 $0 + 0 = 0$

13. $1 \oplus 1 = 0$
 $1 \oplus 0 = 1$
 $0 \oplus 0 = 0$
 $1 \oplus 1 = 0$
 $1 \oplus 0 = 1$
 $0 \oplus 1 = 1$
 $1 \oplus 1 = 0$
 $0 \oplus 0 = 0$

14. 00100101.

Unit 5

**THE 6808 MICROPROCESSOR
PART 1**

CONTENTS

Introduction	5-3
Unit Objectives	5-4
Unit Activity Guide	5-5
Architecture of the 6808 MPU	5-6
Instruction Set of the 6808 MPU	5-15
New Addressing Modes	5-36
Experiments	5-52
Unit Examination	5-53
Examination Answers	5-61

INTRODUCTION

Until now, we have confined our study to a simple hypothetical microprocessor. Obviously, though, this hypothetical model must be very close to the real thing, since we have run its programs on the ET-6808 microprocessor trainer. In this unit you will begin your study of the actual microprocessor upon which our hypothetical model is based. This microprocessor is the 6808.

You already know a great deal about the 6808 microprocessor. Your trainer contains a 6808 microprocessor, so you have been programming this device for the past several units. The main difference between the 6808 microprocessor and our hypothetical model is complexity. As you will see, the 6808 is a vastly expanded version of our hypothetical model.

The 6808 is one of a family of similar devices developed by Motorola over the past few years. The family began with the 6800 and presently includes the 6801, 6802, 6803, 6805, 6808, and 6809. All of these devices use the basic 6800 instruction set which you are learning in this course. Several have additional instructions and addressing modes, on-board memory, and facilities for direct connection to input/output devices. Features such as these are what makes one microprocessor more powerful, more versatile, or more suitable for a particular application than another. For your reference, the data sheets in "Appendix B" give complete specifications on the entire 6800 family.

UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Draw a programming model of the 6808 MPU.
2. Explain the purpose of each block in a simplified block diagram of the 6808 MPU.
3. Using Appendix A and Figure 5-24 as references, explain the operation of all the instructions discussed in this unit.
4. Write simple programs that use indexed and extended addressing.
5. Using Figure 5-24 as a guide, find the opcode, number of MPU cycles, number of bytes, and effects on the condition code flags of every instruction discussed in this unit.

UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on Architecture of the 6808 MPU.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 1 — 8.	_____
<input type="checkbox"/> Read the section on Instruction Set of the 6808 MPU.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 9 — 26.	_____
<input type="checkbox"/> Review “Appendix A.”	_____
<input type="checkbox"/> Read the section on New Addressing Modes.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 27 — 43.	_____
<input type="checkbox"/> Perform Programming Experiments 7 and 8.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

ARCHITECTURE OF THE 6808 MPU

In computer jargon, the word architecture is used to describe the computer's style of construction, its register size and arrangement, its bus configuration, etc. The architecture of our hypothetical microprocessor is shown for one last time in Figure 5-1. By now you should be quite familiar and comfortable with this architecture.

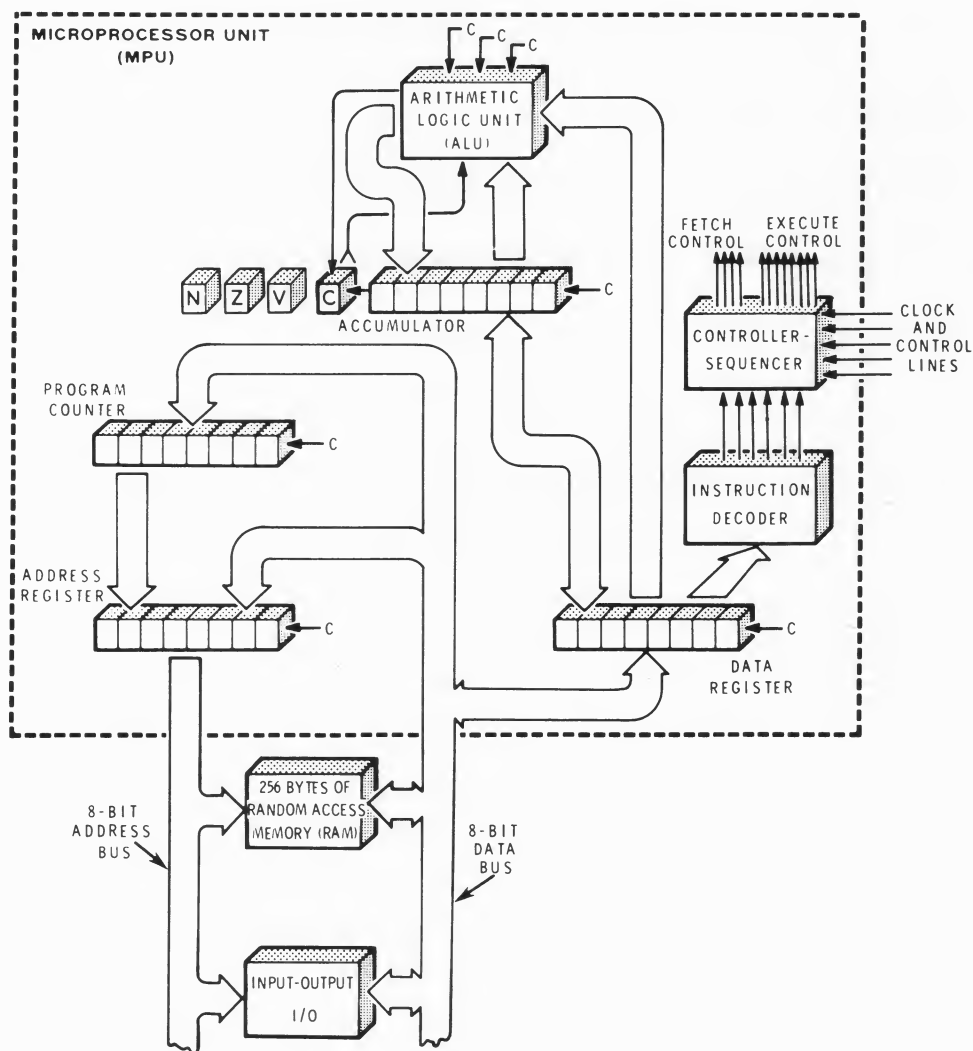


Figure 5-1
Architecture of the hypothetical
microcomputer.

The only reason for showing the details of the model is to give you an idea of what goes on inside the integrated circuit. In an actual microprocessor the internal structure is often so complex that we become bogged down in details if we attempt to analyze it too closely. For this reason, a programming model is generally used when a microprocessor is being introduced for the first time. In the programming model, the emphasis is shifted upward by an order of magnitude. Any register or circuit that cannot be directly controlled by the programmer is simply ignored. Consider the data register for example. There are no instructions that give the programmer direct control over this register. That is, there are no instructions such as Load Data Register, Store Data Register, etc. All data register activity is controlled strictly by the MPU. Thus, the programmer can simply ignore the existence of this register. The same is true of the address register, the instruction decoder, the controller-sequencer, etc. Therefore, the programming model of our hypothetical MPU can be represented as shown in Figure 5-2. This simple diagram is sufficient for most programming applications since it shows all the registers that can be directly controlled by the program.

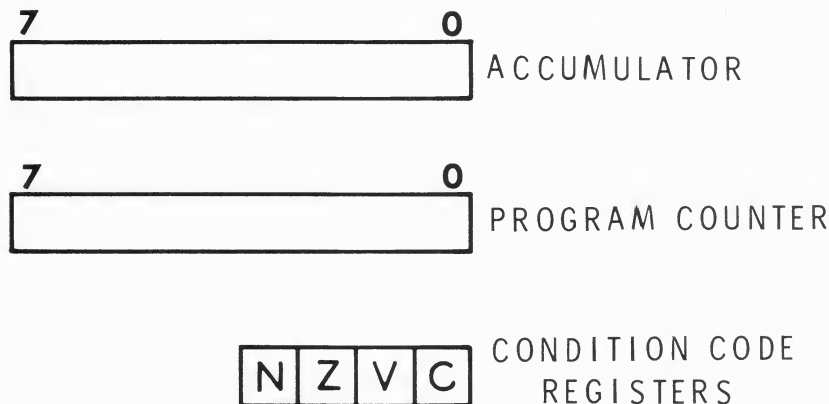
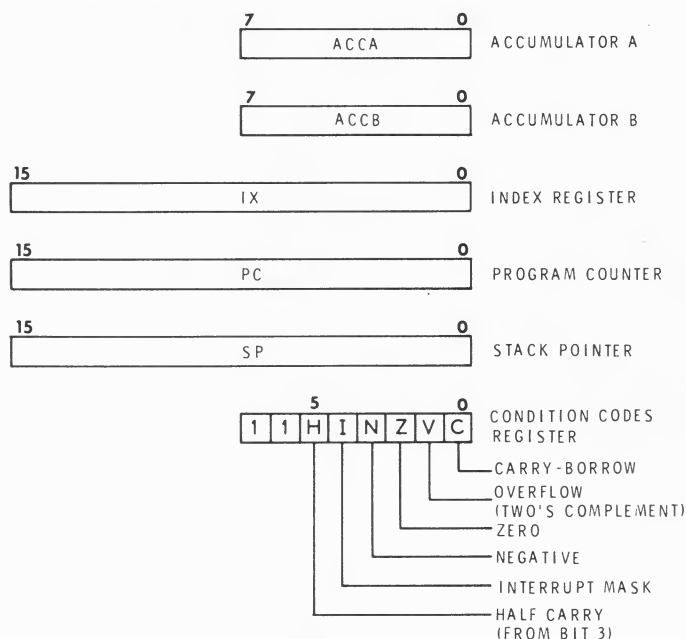


Figure 5-2
Programming model of the
hypothetical MPU.

Programming Model of the 6808 MPU

The 6808 MPU is much more complex than our hypothetical MPU. Consequently, a programming model of the 6808 makes a good starting point. The programming model is shown in Figure 5-3.

Figure 5-3
Programming model of the
6808 MPU.



You will notice immediately that the 6808 MPU has several additional registers. However, only two of these, the index register and the stack pointer, are actually new to you. Let's look at the major differences between this MPU and our hypothetical model.

Two Accumulators The 6808 MPU has two accumulators instead of one. They are called accumulator A (ACCA) and accumulator B (ACCB). Each has its own group of instructions associated with it. The names and mnemonics of the instructions specify which accumulator is to be used. Thus, there are instructions such as:

Load Accumulator A (LDAA)
Load Accumulator B (LDAB)
Store Accumulator A (STAA)
Store Accumulator B (STAB)

Notice that a letter is added to both the name and the mnemonic to indicate which accumulator is being used.

From your previous programming experience, you can visualize the value of a second accumulator. For example consider a program in which the MPU counts the number of times that some operation occurs. In the past, we stored the number that the accumulator was presently working on, loaded the count into the accumulator; incremented the count; stored the count; and reloaded the original number. With a second accumulator, none of this is necessary. We can simply maintain the count in accumulator B while working with the number in accumulator A. In fact, we can perform any arithmetic or logic operation on two different numbers without having to shift the numbers back and forth between memory.

16-Bit Program Counter The program counter in the 6808 has 16_{10} bits rather than 8. Thus, it can specify $65,536_{10}$ different addresses. This means that a 6808 based microcomputer can have up to $65,536_{10}$ bytes of memory. Most applications require substantially less memory than this maximum number. Fortunately, we can use as little or as much memory as we need up to the 2^{16} byte limit.

Since the program counter has 16_{10} bits, the address bus must also be 16-bits wide. Contrast this with the 8-bit address bus of our hypothetical machine.

You may wonder how we specify a 16-bit address with an 8-bit byte. The obvious answer is the two 8-bit bytes are required. Recall that in the direct addressing mode, the address was specified by a single 8-bit byte. The 6808 microprocessor retains this addressing mode. However, since an 8-bit address can specify only 256_{10} addresses, the 6808 MPU can use this mode only if the operand is in the first 256_{10} bytes of memory. To reach higher addresses, a new addressing mode called **extended addressing** must be used. In the extended addressing mode, two bytes are used to represent each address. This addressing mode will be discussed in more detail later. For now, keep in mind that there are $65,536_{10}$ possible addresses. The lowest address is 0000_{16} and the highest is $FFFF_{16}$. Using extended addressing, we have access to any location in memory, but a 2-byte address is required.

Condition Code Registers The 6808 MPU has six condition codes. Four of these are almost identical to those discussed in an earlier unit. These include the negative (N), zero (Z), overflow (V) and carry (C) condition codes. The difference arises because there are two accumulators in the 6808 MPU. Thus, the carry flag is set whenever there is a carry from either accumulator. By the same token, an overflow in either accumulator will set the V flag. Later in this unit, you will see how the condition codes are affected by each instruction.

Two new condition codes are shown in Figure 5-3. The I flag is called an interrupt mask. We will discuss this flag later when you study interrupts. The other is called the half carry flag (H). The H flag is set when there is a carry from bit 3 of the accumulator. The MPU uses this flag to determine how to implement the decimal adjust instruction.

These six flags make up bits 0 through 5 of an 8-bit register. Bits 6 and 7 of the condition code register are not used and are always set to 1. Additional details of the condition codes will be brought out as the need arises.

Index Register The index register is a special-purpose, 16-bit register that greatly increases the power of the microprocessor. It allows a powerful address mode called indexed addressing. We will examine this addressing mode later in this unit. For now, consider the index register to be just another working register. The fact that it holds two bytes instead of one can be put to good use. The MPU has instructions that allow the index register to be loaded from two adjacent memory bytes. Another instruction allows us to store the contents of the index register in two adjacent memory locations. This allows us to move data in 2-byte groups. Also, the index register can be incremented and decremented. This lets us maintain 16-bit tallies.

Stack Pointer The stack pointer is another special-purpose 16-bit register. It allows the MPU and the programmer to use a section of RAM as a last in, first out (LIFO) memory. This capability is extremely valuable when using subroutines or when processing interrupts. These aspects of the stack pointer will be discussed in the next unit. For the time being let's consider the stack pointer to be another 16-bit working register. It too can be loaded from memory, stored in memory, incremented, and decremented.

Block Diagram of the 6808 MPU

Now that you have seen the programming model of the 6808 MPU, take a look at the block diagram. A simplified block diagram is shown in Figure 5-4. Several data paths, most control lines, and a temporary storage register have been omitted in favor of the major data paths and registers.

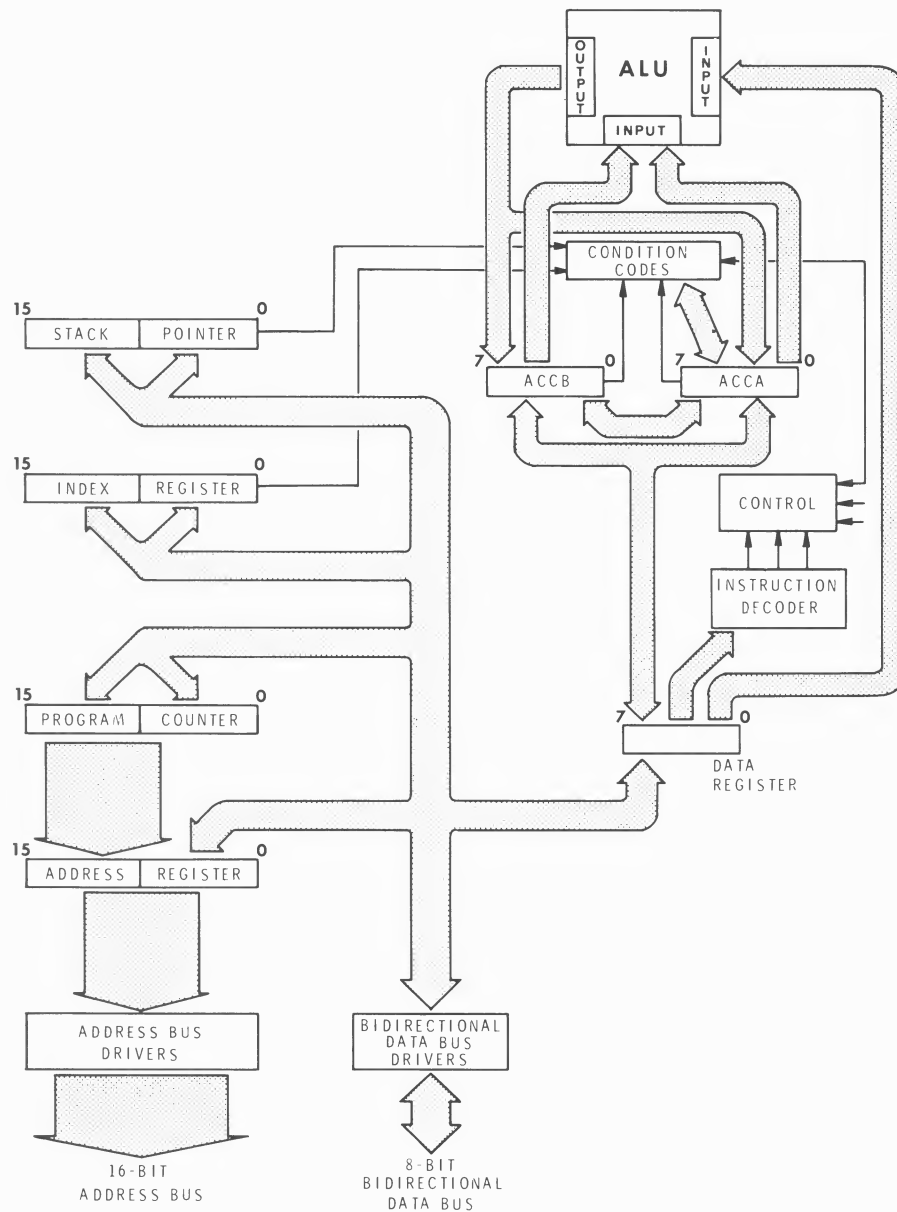


Figure 5-4
Simplified block diagram of the 6808 MPU.

The 16-bit registers are shown on the left. These registers are primarily concerned with addressing memory. Since the address bus has 16-bits, all registers associated with addressing must also have 16-bits. Any of the 16-bit registers can be loaded from the data bus. However, because the data bus has only 8-bits, two operations are required to load the 16-bit registers. The upper half of the affected register is always loaded first. Then, a second operation loads the lower half. Although this requires separate MPU cycles, the microprocessor takes care of these operations automatically. For example, a single instruction can load the 16-bit index register with two memory bytes.

The program counter and address register perform exactly the same functions in the 6808 MPU as they did in our hypothetical model. The fetch and execute phases for the immediate and direct addressing modes are virtually identical. The same is true of the relative addressing mode except that the 8-bit relative address is added to the 16-bit program count.

The 8-bit registers are shown on the right. Notice that these circuits are identical to those in our hypothetical model except that there are two accumulators. The condition code registers monitor both accumulators. Also, the two accumulators share the ALU. This allows you to keep track of two separate mathematical operations at more or less the same time. This arrangement is particularly flexible since the contents of one accumulator can be transferred to the other or the contents of the two accumulators can be added together.

Self-Test Review

1. The microprocessor on which our hypothetical model and the ET-6800 are based is the _____ MPU.
2. A major difference between our hypothetical model and the 6808 MPU is that the latter has two _____.
3. The program counter in the 6808 MPU has _____ bits.
4. How wide is the address bus in a 6808-based microcomputer?
5. What is the range of addresses in the 6808 MPU?
6. List the six condition code flags.
7. Besides the program counter, what other 16-bit registers are used in the 6808 MPU?
8. In the 6808 MPU, does each accumulator have its own carry flag?

Answers

1. 6808.
2. Accumulators.
3. 16_{10} .
4. 16_{10} bits.
5. From 0000 to $65,535_{10}$ or 0000 to $FFFF_{16}$.
6. Carry — borrow (C)
Overflow (V)
Zero (Z)
Negative (N)
Interrupt Mask (I)
Half Carry (H)
7. Index register and stack pointer.
8. No, the two accumulators share a common carry flag.

INSTRUCTION SET OF THE 6808 MPU

The 6808 MPU has about 100_{10} basic instructions. Moreover, when all the different addressing modes are considered, there are 197_{10} different opcodes to which the MPU will respond.

These instructions can be broken down into seven general categories. While some of the categories overlap, the general classifications of instructions are: arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, and condition code. In this unit we will discuss most of these instructions in detail. The handful of instructions that are not discussed in this unit will be described in the following unit.

In this section we will not be concerned with addressing modes. Therefore, no opcodes are given. Later, we will look at the various addressing modes and opcodes. For now, though, let's identify the instructions by their names, mnemonics, and operations. You will see what each instruction does and how it affects the various condition code registers.

Because of the large number of instructions covered in this section, the explanations will be general and brief. You are **not** expected to remember all of the details of every instruction. "Appendix A" of this course contains a detailed listing of each instruction. It explains every detail of the various instructions. After reading this section, turn to "Appendix A" and look over the explanations given there. In the future, when you are in doubt as to exactly what a particular instruction does, look it up in "Appendix A."

Arithmetic Instructions

Figure 5-5 shows the arithmetic instructions of the 6808 MPU. The name of each instruction is given on the left. The next column contains the mnemonics. The center column gives a shorthand description of what the instruction does. The right-hand columns show how the various condition code registers are affected.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Add	ADDA	$A + M \rightarrow A$	↑	●	↑	↑	↑	↑
	ADDB	$B + M \rightarrow B$	↑	●	↑	↑	↑	↑
Add Acmltrs	ABA	$A + B \rightarrow A$	↑	●	↑	↑	↑	↑
Add with Carry	ADCA	$A + M + C \rightarrow A$	↑	●	↑	↑	↑	↑
	ADCB	$B + M + C \rightarrow B$	↑	●	↑	↑	↑	↑
Complement, 2's (Negate)	NEG	$00 - M \rightarrow M$	●	●	↑	↑	①	②
	NEGA	$00 - A \rightarrow A$	●	●	↑	↑	①	②
	NEGB	$00 - B \rightarrow B$	●	●	↑	↑	①	②
Decimal Adjust, A	DAA	Converts Binary Add. of BCD Characters into BCD Format*	●	●	↑	↑	↑	③
Subtract	SUBA	$A - M \rightarrow A$	●	●	↑	↑	↑	↑
	SUBB	$B - M \rightarrow B$	●	●	↑	↑	↑	↑
Subtract Acmltrs.	SBA	$A - B \rightarrow A$	●	●	↑	↑	↑	↑
Subtr. with Carry	SBCA	$A - M - C \rightarrow A$	●	●	↑	↑	↑	↑
	SBCB	$B - M - C \rightarrow B$	●	●	↑	↑	↑	↑

*Used after ABA, ADC, and ADD in BCD arithmetic operation; each 8-bit byte regarded as containing two 4-bit BCD numbers. DAA adds 0110 to lower half-byte if least significant number >1001 or if preceding instruction caused a Half-carry. Adds 0110 to upper half-byte if most significant number >1001 or if preceding instruction caused a Carry. Also adds 0110 to upper half-byte if least significant number >1001 and most significant number = 9.

(Bit set if test is true and cleared otherwise)

- ① (Bit V) Test: Result = 10000000?
- ② (Bit C) Test: Result = 00000000?
- ③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine?
(Not cleared if previously set.)

Figure 5-5
Arithmetic instructions.

To be certain you have the idea, let's go through the first instruction in detail. The first instruction is the add instruction. Actually, since the 6808 has two accumulators, there are two add instructions. Their mnemonics are ADDA and ADDB. Notice that the final letter of the mnemonic indicates which accumulator (A or B) is involved. The shorthand representation of the operation is: $A + M \rightarrow A$. The note at the top of this column tells you that the register labels refer to the contents of the register. Thus, A means the contents of accumulator A and M means the contents of the affected memory location. The symbol (\rightarrow) means "Transfer into." Therefore, $A + M \rightarrow A$ means "Add the contents of accumulator A to the contents of the affected memory location and transfer the sum into accumulator A."

To see how the condition code flags are affected, you simply look over to the right under whatever condition code you are interested in. Generally, the condition code is either unaffected or is tested and set accordingly. When the condition code is unaffected, this is represented by the symbol (\bullet). For example, none of the arithmetic instructions affect the I flag. Most of the arithmetic instructions test the condition codes and set them if the condition exists. For example, if the result of an arithmetic operation is zero, the Z flag is set to 1. If this condition does not exist, the Z flag is reset or cleared to 0. The symbol (\updownarrow) means "test and set if true; clear otherwise." Occasionally, a note is necessary to describe some unusual situation regarding the condition code. This is represented by a number within a circle. The notes are given at the bottom of the drawing.

The ADDA and ADDB instructions are self-explanatory. The ABA instruction adds the contents of accumulator A to the contents of accumulator B. The result is stored in accumulator A.

The add with carry instructions are identical to those discussed earlier for our hypothetical machine. Notice that the carry bit is added in with the sum.

Because two's complement arithmetic is used in the 6800 MPU, instructions are provided that allow us to take the two's complement of a number. The negate instruction subtracts the contents of the affected register from 00_{16} . This is the same as taking the two's complement of the number. The affected register can be any memory location (M) or either accumulator (A or B). Thus, there are three different negate instructions. Keep in mind that NEG means "take the two's complement of the affected memory location;" NEGA means "take the two's complement of accumulator A;" etc.

Notice that the NEG instruction allows us to operate on a byte in memory without first fetching the operand from memory. In the past, we have loaded the operand, performed the operation, and then stored the new operand. However, the 6808 allows us to perform certain operations on the operand without first fetching it from memory. Several examples of this will be pointed out as we progress through the instruction set.

The decimal adjust instruction performs exactly as it did in our hypothetical machine. The note immediately under the table summarizes its operation. It must also be pointed out that this instruction works only with accumulator A.

The subtract and the subtract with carry instructions are self-explanatory. They perform as described earlier for our hypothetical MPU. The 6808 MPU has an additional subtract instruction. The SBA instruction subtracts the contents of accumulator B from the contents of accumulator A. The resulting difference is placed in accumulator A.

Data Handling Instructions

Figure 5-6 shows the largest group of instructions used by the 6808 MPU. These can be loosely categorized as data handling instructions.

The clear instructions allow us to clear a memory location or either accumulator. In the past, we have cleared bytes of memory by first clearing the accumulator and then storing the resulting 00_{16} in the proper memory location. However, the CLR instruction allows us to clear a memory location with a single instruction. Notice that some new entries appear in the condition code registers column. R means that the condition code is always reset or cleared to 0. S means that the code is always set to 1.

The decrement instruction allows us to subtract 1 from a memory location or from either accumulator. The DEC instruction is especially valuable since it allows us to decrement a byte in memory with a single instruction. Previously we have loaded the byte, decremented it, and then stored it back in memory.

The increment instructions are similar except they allow us to add 1 to a memory location or one of the accumulators. Notice that the INC instruction allows us to maintain a tally in memory without having to load it, increment it, and then store it away.

The load accumulator instructions are self-explanatory. Notice that either accumulator can be loaded from memory.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION		COND. CODE REG.					
		(All register labels refer to contents)		5	4	3	2	1	0
OPERATIONS	MNEMONIC			H	I	N	Z	V	C
Clear	CLR	00 → M		●	●	R	S	R	R
	CLRA	00 → A		●	●	R	S	R	R
	CLRB	00 → B		●	●	R	S	R	R
Decrement	DEC	M - 1 → M		●	●	↓	↓	④	●
	DECA	A - 1 → A		●	●	↓	↓	④	●
	DECB	B - 1 → B		●	●	↓	↓	④	●
Increment	INC	M + 1 → M		●	●	↓	↓	⑤	●
	INCA	A + 1 → A		●	●	↓	↓	⑤	●
	INCB	B + 1 → B		●	●	↓	↓	⑤	●
Load Acmltr	LDAA	M → A		●	●	↓	↓	R	●
	LDAB	M → B		●	●	↓	↓	R	●
Rotate Left	ROL	M		●	●	↓	↓	⑥	↓
	ROLA	A		●	●	↓	↓	⑥	↓
	ROLB	B		●	●	↓	↓	⑥	↓
Rotate Right	ROR	M		●	●	↓	↓	⑥	↓
	RORA	A		●	●	↓	↓	⑥	↓
	RORB	B		●	●	↓	↓	⑥	↓
Shift Left, Arithmetic	ASL	M		●	●	↓	↓	⑥	↓
	ASLA	A		●	●	↓	↓	⑥	↓
	ASLB	B		●	●	↓	↓	⑥	↓
Shift Right, Arithmetic	ASR	M		●	●	↓	↓	⑥	↓
	ASRA	A		●	●	↓	↓	⑥	↓
	ASRB	B		●	●	↓	↓	⑥	↓
Shift Right, Logic	LSR	M		●	●	R	↓	⑥	↓
	LSRA	A		●	●	R	↓	⑥	↓
	LSRB	B		●	●	R	↓	⑥	↓
Store Acmltr	STAA	A → M		●	●	↓	↓	R	●
Transfer Acmltrs	STAB	B → M		●	●	↓	↓	R	●
	TAB	A → B		●	●	↓	↓	R	●
	TBA	B → A		●	●	↓	↓	R	●

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of $N \oplus C$ after shift has occurred.

Figure 5-6
Data handling instructions.

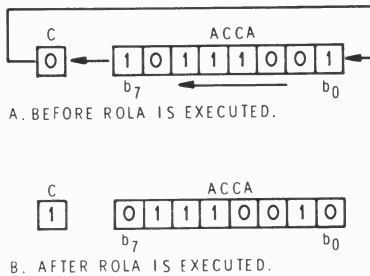


Figure 5-7
Executing the ROLA instruction.

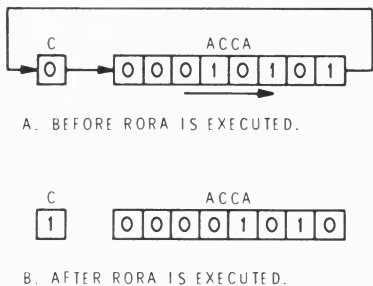


Figure 5-8
Executing the RORA instruction.

The rotate left instructions allow us to shift the contents of the accumulator or a memory location without losing bits of data. Consider the ROLA instruction as an example. When this instruction is executed, the A accumulator and the carry bit form a 9-bit circulating register. That is, they form a closed loop as shown in Figure 5-7A. When ROLA is executed, the data is rotated clockwise. The MSB of A shifts into the carry register. Simultaneously, the contents of A are shifted left. Notice that the carry bit is not lost. Instead it is shifted into the LSB of the accumulator.

While the usefulness of this instruction may not be obvious, it is a valuable tool. For example, it could be used to determine parity. By repeatedly rotating left and testing the C flag, you could determine the number of 1's in the byte. Once you know this, you could easily generate the proper parity bit.

The ROL instruction allows you to rotate a memory byte to the left while it is still in memory. ROLB allows you to rotate the B accumulator to the left. In each case, the C register is used as a ninth bit.

The rotate right instructions are identical except that the direction of rotation is reversed. Figure 5-8 illustrates the execution of the RORA instruction. This instruction is also valuable. Suppose for example that we wish to know if the number in the accumulator is even or odd. This is determined by the LSB of the number. If $LSB = 1$, the number is odd; if $LSB = 0$, the number is even. One way to determine this is to rotate the number to the right so that the LSB is in the C register. We could then test the C register to see if it is set or cleared. Notice that the number could then be restored to its original value by the ROLA instruction.

The arithmetic shift left instruction was discussed earlier in our hypothetical MPU. The ASLA instruction performs exactly as described in the previous unit. However, notice that the 6808 MPU also has an ASLB instruction that performs the same operation with accumulator B. Also, it has an ASL instruction that allows us to perform this operation on a byte that is in memory. Figure 5-9 illustrates the execution of this instruction.

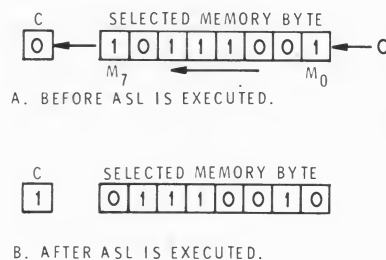


Figure 5-9
Executing the ASL instruction.

While there is only one type of shift left instruction, there are two types of shift right instructions. Let's discuss the arithmetic shift right instructions first.

When an **arithmetic** shift right instruction is executed, the number in the affected register is shifted right one position. The LSB goes into the C register. B_1 shifts to B_0 , etc. B_7 shifts into B_6 . However, B_7 itself remains unchanged. Figure 5-10 illustrates the execution of the ASRB instruction. Notice that there are also ASRA and ASR instructions listed in Figure 5-6. These perform the same type of shift operation but on accumulator A and the selected memory byte respectively.

The logic shift right instructions are different in that they do not retain the sign bit. When a logic shift right is executed, the contents of the affected register are shifted to the right. The LSB goes into the carry register. The MSB is filled with a 0. For example, the LSR instruction is illustrated in Figure 5-11. While this instruction shifts the selected memory locations, LSRA and LSRB can be used to perform similar operations on accumulators A and B respectively.

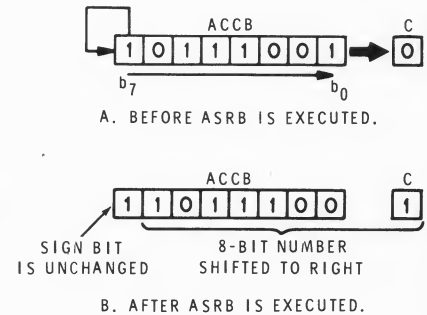


Figure 5-10
Executing the ASRB instruction.

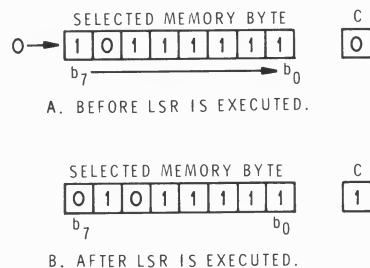


Figure 5-11
Executing the LSR instruction.

Referring back to Figure 5-6, the store accumulator instructions are self-explanatory.

The final data handling instructions are the transfer accumulator instructions. TAB copies the contents of accumulator A into accumulator B. After this instruction is executed, the number originally in accumulator A will be in both accumulators. TBA does just the opposite. It copies the contents of accumulator B into accumulator A. After TBA is executed, the number originally in accumulator B will be in both accumulators.

Logic Instructions

The logic instructions in the 6808 MPU are similar to those in our hypothetical MPU. Figure 5-12 shows the 6808's logic instructions.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
And	ANDA	$A \bullet M \rightarrow A$	●	●	↓	↓	R	●
	ANDB	$B \bullet M \rightarrow B$	●	●	↓	↓	R	●
Complement, 1's	COM	$\overline{M} \rightarrow M$	●	●	↓	↓	R	S
	COMA	$\overline{A} \rightarrow A$	●	●	↓	↓	R	S
	COMB	$\overline{B} \rightarrow B$	●	●	↓	↓	R	S
Exclusive OR	EORA	$A \oplus M \rightarrow A$	●	●	↓	↓	R	●
	EORB	$B \oplus M \rightarrow B$	●	●	↓	↓	R	●
Or, Inclusive	ORA	$A + M \rightarrow A$	●	●	↓	↓	R	●
	ORB	$B + M \rightarrow B$	●	●	↓	↓	R	●

Figure 5-12
Logic instructions.

There is one AND instruction for each accumulator. The contents of the specified accumulator are ANDed bit-for-bit with the contents of the selected memory location. The result is placed back in the accumulator. This is identical to the AND instruction in our hypothetical machine.

The complement instructions allow you to take the 1's complement of the number in the affected register. COM allows you to complement a byte in memory.

COMA and COMB allow you to complement the contents of accumulators A and B respectively. In each case, all 1's are changed to 0's and all 0's are changed to 1's.

The exclusive OR instructions work like the one in our hypothetical MPU. The contents of the specified accumulator are exclusively ORed bit-for-bit with the contents of the selected memory location. The result is stored back in the specified accumulator.

The inclusive OR is similar except that the contents of the specified accumulator are inclusively ORed with the contents of the selected memory location.

Data Test Instructions

These are a powerful group of instructions that allow us to compare operands in several different ways. In previous units, you had experience comparing operands. The most frequently used method was to subtract one operand from another and test the result for zero or negative. In many cases, the numeric result of the subtraction was unimportant. We needed to know only if the result was zero or minus. The data test instructions allow us to make several different tests without actually producing an unwanted numeric result. These instructions are shown in Figure 5-13.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Bit Test	BITA	A • M	•	•	↓	↓	R	•
	BITB	B • M	•	•	↓	↓	R	•
Compare	CPMA	A – M	•	•	↓	↓	↓	↓
	CMPB	B – M	•	•	↓	↓	↓	↓
Compare Acmltrs	CBA	A – B	•	•	↓	↓	↓	↓
Test, Zero or Minus	TST	M – 00	•	•	↓	↓	R	R
	TSTA	A – 00	•	•	↓	↓	R	R
	TSTB	B – 00	•	•	↓	↓	R	R

Figure 5-13

Data test instructions.

The bit test instructions are very similar to the AND instructions. In both cases, the contents of the specified accumulator are ANDed with the contents of the selected memory location. The difference is that with the bit test instruction no logical product is produced. Neither the contents of the accumulator nor memory are altered in any way. However, the condition code registers are affected just as if the AND operation had taken place. Consider the BITA instruction. When executed, A is ANDed with M. If the result is 00₁₆, the Z register is set. Otherwise, the Z register is cleared. If the MSB of the result is 1, the N flag is set. However, the contents of the accumulator and memory are unaffected.

In the same way, the compare instructions are similar to subtract instructions except that the resulting numeric difference is ignored. For example, when the CPMA instruction is executed, the contents of the selected memory location are subtracted from the contents of accumulator A. The condition codes are affected just as if a difference had been produced. However, the original contents of accumulator A and memory are unaffected.

The compare accumulators instruction (CBA) works the same way. The condition codes are set as if the contents of B were subtracted from the contents A. However, the contents of the accumulators are unaffected.

Finally, the test for zero or minus instruction allows you to test the number in one of the accumulators or the memory to see if it is negative or zero. When this instruction is executed, the MPU looks at the number in question and sets the N and Z flags accordingly. The number itself is not changed.

Index Register and Stack Pointer Instructions

The index register and stack pointer are 16-bit registers. Figure 5-14 shows eleven instructions that allow us to control the operation of these registers. Because of the 16-bit format, the load, store, and compare instructions are slightly different from those discussed earlier.

INDEX REGISTER AND STACK			5	4	3	2	1	0
POINTER OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION	H	I	N	Z	V	C
Compare Index Reg	CPX	$(X_H/X_L) - (M/M + 1)$	•	•	①	↓	②	•
Decrement Index Reg	DEX	$X - 1 \rightarrow X$	•	•	•	↓	•	•
Decrement Stack Pntr	DES	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX	$X + 1 \rightarrow X$	•	•	•	↓	•	•
Increment Stack Pntr	INS	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	③	↓	R	•
Load Stack Pntr	LDS	$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	③	↓	R	•
Store Index Reg	STX	$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	③	↓	R	•
Store Stack Pntr	STS	$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	③	↓	R	•
Indx Reg \rightarrow Stack Pntr	TXS	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr \rightarrow Indx Reg	TSX	$SP + 1 \rightarrow X$	•	•	•	•	•	•

- ① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?
 ② (Bit V) Test: 2's complement overflow from subtraction of LS bytes?
 ③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Figure 5-14
Index register and stack pointer
instructions.

The compare index register (CPX) instruction allows us to compare the 16-bit number in the index register with any two consecutive bytes in memory. Recall that the index register (X) will hold two bytes. The higher byte is identified as X_H while the lower byte is called X_L . When the CPX instruction is executed, X_H is compared with the 8-bit byte in the specified memory location (M). Also, X_L is compared with the byte immediately following the specified memory location (M+1). The comparison is the same as if M and M+1 were subtracted from X_H and X_L except that no numeric difference is produced. Neither X nor M is changed in any way. However, the N, Z, and V condition codes are affected as shown in Figure 5-14. Generally, the Z code is the one we are interested in since it tells us whether or not an exact match exists between the index register and the two bytes in memory.

The next four instructions are self-explanatory. They allow us to increment and decrement either the index register or the stack pointer. For one thing, these instructions allow us to maintain two separate 16-bit tallies simultaneously. However, the real value of these instructions and their associated registers will be discussed later.

The load and store instructions for the 16-bit registers are shown next in Figure 5-14. Since these are two byte registers, the LDX and LDS instructions must load two bytes from memory. In the case of the index register, the specified memory byte (M) is loaded into the upper half of the index register (X_H). An instant later, the next byte in memory (M+1) is automatically loaded into the lower half of the index register (X_L). Thus, the operation can be described as: $M \rightarrow X_H, (M+1) \rightarrow X_L$.

Because the stack pointer is also a 16-bit register, the load stack pointer instruction (LDS) works the same way. Its operation can be described as: $M \rightarrow SP_H, (M+1) \rightarrow SP_L$. Here, SP_H refers to the upper half of the stack pointer while SP_L refers to the lower half.

When the contents of the 16-bit registers are being stored, the operation is reversed. For example, the STX instruction stores X_H in M and X_L in M+1. A similar instruction, STS, allows us to store the contents of the stack pointer in the same way.

The final two instructions in this group allow us to transfer numbers between these two 16-bit registers. The TXS instruction loads the stack pointer with the contents of the index register **minus one**. The TSX instruction loads the index register with the contents of the stack pointer **plus one**. A more detailed discussion of these two important registers and their associated instructions will be given in the next unit.

Branch Instructions

The branch instructions are shown in Figure 5-15. Two additional instructions are also thrown in since they affect the program counter.

BRANCH OPERATIONS	MNEMONIC	BRANCH TEST	5	4	3	2	1	0
			H	I	N	Z	V	C
Branch Always	BRA	None	•	•	•	•	•	•
Branch If Carry Clear	BCC	$C = 0$	•	•	•	•	•	•
Branch If Carry Set	BCS	$C = 1$	•	•	•	•	•	•
Branch If = Zero	BEQ	$Z = 1$	•	•	•	•	•	•
Branch If \geq Zero	BGE	$N \oplus V = 0$	•	•	•	•	•	•
Branch If $>$ Zero	BGT	$Z + (N \oplus V) = 0$	•	•	•	•	•	•
Branch If Higher	BHI	$C + Z = 0$	•	•	•	•	•	•
Branch If \leq Zero	BLE	$Z + (N \oplus V) = 1$	•	•	•	•	•	•
Branch If Lower Or Same	BLS	$C + Z = 1$	•	•	•	•	•	•
Branch If $<$ Zero	BLT	$N \oplus V = 1$	•	•	•	•	•	•
Branch If Minus	BMI	$N = 1$	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	$Z = 0$	•	•	•	•	•	•
Branch If Overflow Clear	BVC	$V = 0$	•	•	•	•	•	•
Branch If Overflow Set	BVS	$V = 1$	•	•	•	•	•	•
Branch If Plus	BPL	$N = 0$	•	•	•	•	•	•
No Operation	NOP	Advances Prog. Cntr. Only	•	•	•	•	•	•
Wait for Interrupt	WAI		•	①	•	•	•	•

- ① (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Figure 5-15
Jump and branch instructions.

Nine of these instructions were discussed in the previous unit. These are: Branch Always (BRA); Branch If Carry Clear (BCC); Branch If Carry Set (BCS); Branch If Equal Zero (BEQ); Branch If Not Equal Zero (BNE); Branch If Minus (BMI); Branch If Plus (BPL); Branch If Overflow Clear (BVC); and Branch If Overflow Set (BVS).

Before we discuss the new branch instructions, here are some of the symbols we will be using. The symbol (\geq) means “is greater than or is equal to”; $(>)$ means “is greater than”; (\leq) means “is less than or is equal to”; $(<)$ means “is less than”; and (\neq) means “is not equal to.”

Now consider the Branch If Greater Than or Equal instruction (BGE). This instruction is normally used after a subtract or compare instruction. It will cause a branch operation if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory. This condition is indicated by the N and V flags having the same value. The MPU determines if this condition is met by exclusively ORing N and V and examining the result.

Three simple examples may help illustrate the operation of this instruction. Let's start with a number in the accumulator that is greater than the operand in memory:

Number in Accumulator	=	00000010 ₂
Operand in Memory	=	00000001 ₂

When the operand is subtracted, the result is 00000001₂. With this result, both N and V are cleared to 0. Notice that N and V are equal and $N \oplus V = 0$. If the BGE instruction followed the subtract operation, the branch would be implemented.

Now see what happens when the number in the accumulator is equal to the operand:

Number in Accumulator	=	00000010 ₂
Operand in Memory	=	00000010 ₂

When the operand is subtracted, the result is 00000000₂. Again N and V are cleared to 0. Thus, N and V are still equal and $N \oplus V = 0$. Again, the BGE instruction would cause a branch to occur.

Finally, note what happens when the number in the accumulator is smaller:

Number in Accumulator	=	00000001 ₂
Operand in Memory	=	00000010 ₂

When the operand is subtracted, the result is 11111111₂. This time N is set but V is cleared. Thus, N and V are not equal. Therefore, $N \oplus V = 1$. In this case, the BGE conditions are not met and no branch will occur. The branch occurs if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory.

Next, consider the Branch If Greater Than (BGT) instruction. This instruction is normally used immediately after a subtract or compare operation. The branch will occur only if the two's complement minuend was greater than the two's complement subtrahend. By trying several examples as we did above, you will find that the branch conditions are met when $Z = 0$ and $N = V$.

The Branch If Higher (BHI) instruction is similar to the BGT instruction except that it is concerned with **unsigned** numbers. BHI is normally used after a subtract or compare operation. The branch will occur only if the unsigned minuend was greater than the unsigned subtrahend. By trying several different examples, you can prove that this occurs only when the C and Z flags are both 0.

The Branch If Less Than or Equal (BLE) instruction allows you to compare two's complement numbers in another way. If it is executed immediately after a subtract or compare operation, the branch will occur only if the two's complement minuend was less than or equal to the two's complement subtrahend.

The Branch If Lower Or Same (BLS) instruction is similar to the BLE instruction except that **unsigned** numbers are compared. When it is executed immediately after a subtract or compare operation, the branch will occur only if the unsigned minuend was lower than or equal to the unsigned subtrahend.

The Branch If Less Than Zero (BLT) instruction is also similar to the BLE instruction except that the equal qualification is removed. If BLT is executed immediately after a subtract or compare operation, the branch occurs only if the two's complement minuend was less than the two's complement subtrahend.

Two additional instructions are included in Figure 5-15. Although they are not branch instructions, they are included here since they do not seem to fit any of the other categories.

The No Operation (NOP) instruction is a “do-nothing” instruction that simply consumes a small increment of time. It does not change the contents of any register except the program counter. It does increment the program counter by one and consumes two MPU cycles. In spite of this, the NOP is a very useful instruction. When writing a program, we frequently use too many instructions. Once the program is loaded in memory, it is often inconvenient to simply remove an instruction. The hole left in memory can be filled by moving back all instructions that follow. However, a faster way is to simply fill the hole with one or more NOP instructions.

The Wait For Interrupt (WAI) instruction is the 6808's version of a HLT instruction. In earlier units we used this instruction at the end of all our programs. We will continue to use it in the same manner in the future. However, as you will see in the next unit, there is more involved in executing the WAI instruction than simply stopping the MPU. For now, though, continue to think of the WAI as a simple halt instruction.

Condition Code Register Instructions

The 6808 MPU has eight instructions that allow us direct access to the condition codes. These are listed in Figure 5-16.

CONDITION CODE REGISTER			5	4	3	2	1	0
OPERATIONS	MNEMONIC	BOOLEAN OPERATION	H	I	N	Z	V	C
Clear Carry	CLC	0 → C	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0 → I	•	R	•	•	•	•
Clear Overflow	CLV	0 → V	•	•	•	•	R	•
Set Carry	SEC	1 → C	•	•	•	•	•	S
Set Interrupt Mask	SEI	1 → I	•	S	•	•	•	•
Set Overflow	SEV	1 → V	•	•	•	•	S	•
Accmltr A → CCR	TAP	A → CCR	①					
CCR → Accmltr A	TPA	CCR → A	•	•	•	•	•	•

R = Reset

S = Set

• = Not affected

① (ALL) Set according to the contents of Accumulator A.

Figure 5-16

Condition code register instructions.

Most of these instructions are self-explanatory. The Clear Carry (CLC) instruction resets the C flag to 0 while the Set Carry (SEC) sets it to 1. In the same way, the CLV and SEV instructions allow us to clear and set the overflow flag. Also, the CLI and SEI instructions can be used to clear or set the interrupt flag.

You will notice that there are no instructions for individually clearing the N, Z, or H flags. However, we can still set or clear these flags with the Transfer Accumulator A to the Processor Status Register (TAP) instruction. Figure 5-17 illustrates the execution of this instruction. The contents of bits 0 through 5 of accumulator A are transferred to the condition code registers. Thus, this instruction allows us to set or clear all the condition codes at once.

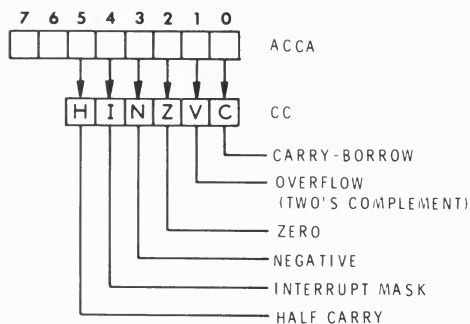


Figure 5-17

Executing the TAP instruction.

The final instruction is the Transfer Processor Status to Accumulator A (TPA) instruction. When this instruction is executed, the contents of the condition code registers are transferred to bits 0 through 5 of accumulator A. This operation is illustrated in Figure 5-18. Notice that bits 6 and 7 of the accumulator are set to 1.

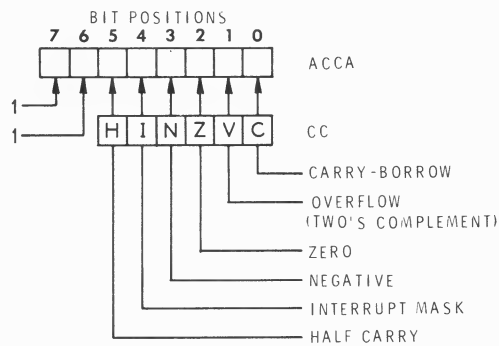


Figure 5-18
Executing the TPA instruction.

Summary of Instruction Set

As you can see, the 6808 MPU has a wide variety of instructions. In this section, most of the instructions have been mentioned briefly. However, a full explanation of some instructions must wait until additional new concepts have been covered.

In one short section, it is very difficult to cover every instruction in detail. And, it is virtually impossible for the reader to remember all the details of each instruction. Remember, all of the instructions available to the 6808 MPU are explained in detail in Appendix A of this program. Also, they are arranged alphabetically by their mnemonics for easy reference. Refer to Appendix A any time you are in doubt about what an instruction does. Be sure to look over the introductory material in the Appendix so that you understand all the conventions and symbols.

Self-Test Review

9. List the seven general categories of instructions.
10. What is meant by the shorthand notation: $A+B \rightarrow A$?
11. How is the C flag affected by the “add” and “add with carry” instructions?
12. Is the C flag changed when the AND instruction is executed?
13. Explain the difference between the NEG instruction and the COM instruction.
14. Explain the difference between the ANDA instruction and the BITA instruction.
15. The decimal adjust instruction is associated with which accumulator?
16. When the RORA instruction is executed the LSB of accumulator A is shifted into the _____ register.
17. List eleven operations that can be performed directly to an operand in memory without first loading it into one of the MPU registers.

18. Explain the difference between the SUBB instruction and the CMPB instruction.
19. List the four types of logic operations that the 6800 MPU can perform.
20. When the LDX instruction is executed, from where is the index register loaded?
21. List four conditional branch instructions that are commonly used after a compare or subtract instruction to compare two's complement numbers.
22. Explain the difference between the BGT and BHI instructions.
23. Which instruction is often used to fill in a hole left in a program after an unwanted byte is removed?
24. Which instruction in the 6808 roughly corresponds to the halt instruction in our hypothetical machine?
25. Which of the condition codes can be individually set or cleared?
26. When you have some doubt as to exactly what operation is performed by a given instruction, where can you look to find the answer?

Answers

9. Arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, condition code.
10. Add the contents of accumulator A to the contents of accumulator B; transfer the result to accumulator A.
11. The C flag is set if a carry occurs; it is cleared otherwise.
12. No, the C flag is unaffected by the AND instruction.
13. The COM instruction replaces the operand with its 1's complement. The NEG instruction replaces the operand with its 2's complement.
14. With the ANDA instruction, the result of the AND operation is placed in accumulator A. With the BITA instruction, the condition code registers are set according to the result but the result is not retained.
15. The decimal adjust instruction works only with the A accumulator.
16. Carry (C).
17. A byte in memory can be: cleared, incremented, decremented, complemented, negated, rotated left, rotated right, shifted left arithmetically, shifted right arithmetically, shifted right logically, and tested.

18. With the SUBB instruction, a difference is produced and placed in accumulator B. With CMPB, the flags are set as if a difference were produced, but the difference is not retained.
19. Complement, AND, inclusive OR, and exclusive OR.
20. The upper half of the index register is loaded from the specified memory location; the lower half from the byte following the specified memory location.
21. BGE, BGT, BLE, BLT.
22. BGT is used to test the result of subtracting two's complement numbers. BHI is used to test the result of subtracting unsigned numbers.
23. NOP.
24. WAI.
25. C, I, and V.
26. Appendix A of this course.

NEW ADDRESSING MODES

In previous units, we have discussed four addressing modes. Let's briefly review these.

In the immediate addressing mode, the operand is the memory byte immediately following the opcode. These are generally two byte instructions. The first byte is the opcode, the second is the operand. However, there are exceptions to the two-byte rule. Some operations involve the 16-bit index register and stack pointer. In these cases, the operand is the **two** bytes immediately following the opcode. These are three byte instructions. The first byte is the opcode, the second and third are the operand.

In the direct addressing mode, the byte following the opcode is the address of the operand. These are always two byte instructions. The first byte is the opcode; the second is the address of the operand. An eight-bit byte can specify addresses from 00 to FF_{16} . Thus, when the direct addressing mode is being used, the operand must be in the first 256_{10} bytes of memory. Since the 6808 MPU can have up to $65,536_{10}$ bytes of memory, another means must be used to address the upper portion of memory.

The relative addressing mode is used for branching. These are two byte instructions. The first byte is the opcode, the second is the relative address. Recall that the relative address is added to the program count to form the absolute address. Since the 8-bit relative address is a two's complement number, the branch limits are $+127_{10}$ and -128_{10} .

In the inherent addressing mode either there is no operand or the operand is implied by the instruction. These are one byte instructions.

In this section, we will discuss two new addressing modes. These are called **extended addressing** and **indexed addressing**. We will discuss extended addressing first.

Extended Addressing

Extended addressing is similar to direct addressing but with one significant difference. Recall that with direct addressing the operand must be in the first 256_{10} bytes of memory. Since this represents less than one percent of the addresses available to the 6808 MPU, a more powerful addressing mode is needed. The extended addressing mode fills this need.

The format of an instruction that uses extended addressing is shown in Figure 5-19. The instruction will always have three bytes. The first byte is the opcode. The second and third bytes form a 16-bit address. Notice that the most significant part of the address is the byte immediately following the opcode. Since this instruction has a 16-bit address, the operand can be at any one of the $65,536_{10}$ possible addresses.

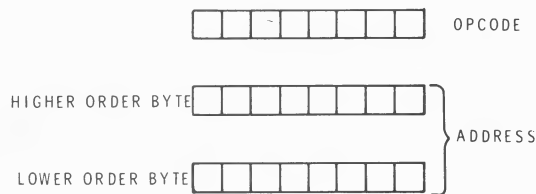


Figure 5-19

Format of an instruction that uses the
extended addressing mode.

Suppose, for example, that you wish to load the operand at memory location 2134_{16} into accumulator B. The instruction would look like this:

F6	Opcode for LDAB extended
21	Higher order address
34	Lower order address

By the same token, if you wish to increment the number in memory location $AA00_{16}$, the instruction would be:

7C	Opcode for INC extended
AA	Higher order address
00	Lower order address

The extended addressing mode allows us to address an operand at any address including the first 256_{10} bytes of memory. Thus, if you wish to load the operand at address 0013_{16} into accumulator A, you can use extended addressing:

B6	Opcode for LDAA extended
00	Higher order address
13	Lower order address

Or, you can use direct addressing:

96	Opcode for LDAA direct
13	Address

Notice that, with direct addressing, the higher order address can be ignored since it is always 00. Because it saves one memory byte and one MPU cycle, direct addressing is normally used when the operand is in the first 256_{10} bytes of memory. Extended addressing is used when the operand is above address $00FF_{16}$. However, as you will see later, some instructions do not have a direct addressing mode. In these cases, extended addressing must be used even if the operand is in the first 256_{10} memory locations.

Indexed Addressing

The most powerful mode available to the 6808 MPU is indexed addressing. Recall that the 6808 MPU has a 16-bit index register. There are several instructions associated with this register. They allow us to load the register from memory and to store its contents in memory. Also, we can increment and decrement the index register. We can even compare its contents with two consecutive bytes in memory. These capabilities alone make the index register a very handy 16-bit counter. However, the real power of the index register comes from the fact that we can use this counter as an address pointer. Since this is a 16-bit register, it can point to any address in memory.

Purpose Before going into the details of how indexed addressing works, let's see why it is needed. Let's assume that we wish to add a list of 20_{16} numbers, and that the numbers are in 20_{16} consecutive memory locations starting at address 0050. Using the addressing modes discussed earlier, our program might look like this:

CLRA	Clear Accumulator A.
ADDA	Add the first number
50	To accumulator A.
ADDA	Add the second
51	number to accumulator A.
ADDA	Add the third number
52	to accumulator A.
.	.
.	.
.	.
ADDA	Add the last number
6F	to accumulator A.
WAI	Wait.

While this accomplishes the desired result, it requires a long repetitive program. The above program would require 66_{10} bytes of memory. Notice that all the ADDA instructions are identical except that each successive address is one larger than the previous address. Indexed addressing can greatly simplify programs of this type.

Instruction Format The format of an instruction that uses indexed addressing is shown in Figure 5-20. Notice that this is a two-byte instruction. The first byte is the opcode, and the second is called an offset address. The offset address is an **unsigned** 8-bit binary number. It is added to the contents of the index register to determine the address at which the operand is located.

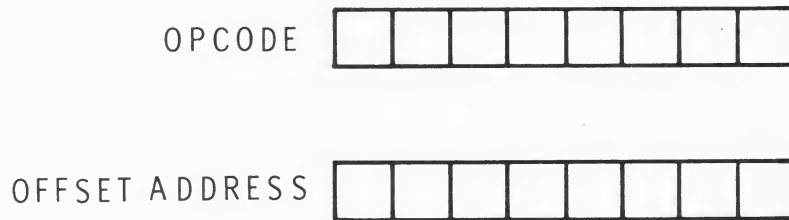


Figure 5-20

Format of an instruction that uses
the indexed addressing mode.

Every instruction that involves an operand in memory can use the indexed addressing mode. In this unit, we will use the following convention to indicate indexed addressing:

LDAA, X
STAA, X
ADDB, X
etc.

In each case, the X tells us that indexed addressing is used. For example, the first instruction means: “using indexed addressing, load the contents of the specified memory location into accumulator A.” Now let’s see how the address of the operand is determined.

Determining the Operand Address When indexed addressing is being used, the address of the operand is determined by the offset address and the number in the index register. Specifically, the 8-bit offset address is added to the 16-bit address in the index register. The 16-bit sum becomes the address of the operand. Figure 5-21 illustrates this.

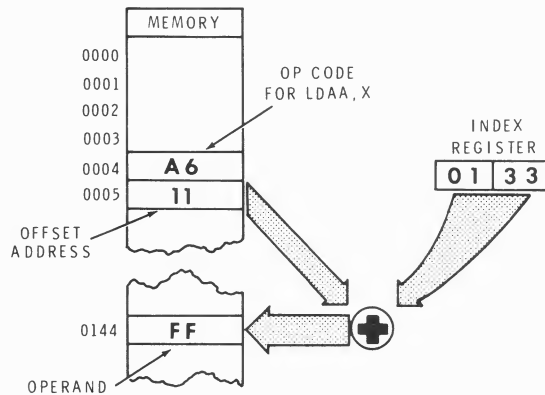


Figure 5-21

The operand address is formed by adding the offset address to the contents of the index register.

Here, the instruction in memory location 0004_{16} is LDAA, X. The offset address is 11_{16} . The contents of the index register are 0133_{16} . When the LDAA, X instruction is executed, the address of the operand is formed by adding the offset address to the number in the index register. In this case, the operand address will be:

$$\begin{array}{r} 0133_{16} \\ + \quad 11_{16} \\ \hline 0144_{16} \end{array}$$

The operand at this address is loaded into accumulator A. In this example, the operand FF is loaded into accumulator A when the instruction at location 0004 is executed. It is important to remember that this does not change the contents of the index register in any way. That is, the index register will still contain 0133_{16} after the instruction is executed.

Adding a List of Numbers To see how this addressing mode saves instructions, consider the problem given earlier. Recall that we were to add 20_{16} numbers stored in consecutive memory locations starting at address 0050. Using indexed addressing for the add instruction, our program looks like the one shown in Figure 5-22.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with the
0011	00	00	address of the
0012	50	50	first number in list.
0013	4F	CLRA	Clear accumulator A
0014	AB	→ ADDA, X	Add to accumulator A using indexed addressing
0015	00	00	with an offset address of 00.
0016	08	INX	Increment index register.
0017	8C	CPX #	Compare the contents of the index register
0018	00	00	with an address that is one greater than the
0019	70	70	address of the last number in the list.
001A	26	BNE	If not equal, branch back
001B	F8	→ F8	to the ADDA, X instruction.
001C	3E	WAI	Otherwise, halt.

Figure 5-22

Program for adding a list of
 20_{16} numbers.

The first instruction is load index register immediate. Notice that a new symbol is used in this program. The symbol # is used to indicate the immediate addressing mode. Thus, the LDX# instruction causes the operand immediately following the opcode to be loaded into the index register. Recall that the index register can hold two 8-bit bytes. The operand is the two-byte number 0050_{16} . You may recognize that this is the address of the first number in the list of numbers that is to be added.

The next instruction clears accumulator A. The sum will be accumulated in this register, so it is important that it be cleared initially.

The third instruction (ADDA, X) is the only instruction in the program that uses indexed addressing. Notice that the symbol X indicates the indexed addressing mode. The offset address is 00. Recall that the operand address is determined by adding the offset to the contents of the index register. The index register contains 0050_{16} from a previous instruction. Since the offset is 00, the operand address is 0050_{16} . That is, the contents of memory location 0050 are added to the contents of accumulator A. Recall that 0050_{16} is the address of the first number in the list.

The fourth instruction increments the index register to 0051₁₆. Notice that the index register now points to the address of the second number in the list.

The fifth instruction compares the number in the index register with a number that is one greater than the address of the last number in the list.

If a match occurs, the Z flag will be set. Of course in this case, no match occurs yet. Notice once again that the symbol # indicates the immediate addressing mode. Thus, the contents of the index register are compared with the next two bytes in the program or 0070.

The BNE instruction tests the Z flag to see if the two numbers matched. If no match is indicated, the relative address (F8) directs the program back to the ADDA, X instruction. The first pass through the loop ends with the first number in accumulator A.

The second pass through the loop begins with the ADDA, X instruction being executed again. This time the index register points to address 0051. Therefore, the second number in the list is added to accumulator A. Accumulator A now contains the sum of the first two numbers. The index register is then incremented to 0052. Its contents are again compared with 0070. No match exists so the BNE instruction causes the loop to be repeated again.

The loop is repeated over and over again. Each time, the next number in the list is added to accumulator A. This process continues until the last number in the list is added. At that time, the index register will be incremented to 0070. Thus, when the CPX# instruction is executed, the Z flag will be set because the two numbers match. The BNE instruction recognizes that a match has occurred. Consequently, it does not allow the branch to occur and the next instruction in sequence is executed. Because this is the WAI instruction, the program halts. At this time, the sum of the 20₁₆ numbers in the list will be in accumulator A.

Adding a list of numbers is a classic example of how indexing can be used to shorten a program. However, this example does not illustrate the full power of indexed addressing. For example, it does not illustrate the advantage of the offset address. Because indexed addressing is so important, let's look at another example.

Copying a List Let's assume we have a list of 10_{16} numbers that we wish to copy from one location to another. For simplicity, assume that the list is presently in addresses 0030 through 003F and that we wish to copy the list in location 0040 through 004F. Without using indexed addressing, our program might look like this:

```
LDAA
 30
STAA
 40
LDAA
 31
STAA
 41
.
.
.
LDAA
 3F
STAA
 4F
WAI
```

As you have seen; long, repetitive programs such as this are excellent candidates for indexed addressing.

Using indexed addressing, our program might look like that shown in Figure 5-23. The first step is to load the index register with the first address in the original list. The LDAA, X instruction has an offset address of 00. Therefore, accumulator A is loaded from the address specified by the index register (0030). That is, the first number in the original list is loaded into accumulator A when the LDAA, X instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with
0011	00	00	the first address of the original
0012	30	30	list.
0013	A6	LDAA, X	Load accumulator A indexed with
0014	00	00	an offset of 00.
0015	A7	STAA, X	Store accumulator A indexed with
0016	10	10	an offset of 10 ₁₆ .
0017	08	INX	Increment index register.
0018	8C	CPX #	Compare index with one greater
0019	00	00	than last
001A	40	40	address in original list.
001B	26	BNE	If not equal, branch back to the
001C	F6	F6	LDAA, X instruction.
001D	3E	WAI	Otherwise, halt.

Figure 5-23

Program for copying a list from
addresses 0030 — 003F into
addresses 0040 — 004F.

The STAA, X instruction illustrates the use of the offset address. Notice that the offset is 10. This number is added to the address in the index register to form the effective address at which the contents of accumulator A are stored. Thus, the contents of accumulator A are stored at address 0040. Remember, this does not change the number in the index register in any way. By using the offset, we can load the accumulator indexed from one address and store the accumulator indexed at another.

Next, the index register is incremented to 0031. It is then compared with 0040. Since no match exists, the BNE instruction directs the program back to the LDAA, X instruction. The loop is repeated until the entire list is rewritten in locations 0040 through 004F. After the last entry in the list is copied, the index register is incremented to 0040. Thus, the CPX# instruction sets the Z flag allowing the BNE instruction to divert the program from the loop. The program halts after the last entry in the list is written in its new position in memory.

Instruction Set Summary

You have now been introduced to most of the instructions available to the 6808 MPU. You have also been introduced to all of the addressing modes. Now let's look at the complete instruction set.

Figure 5-24 summarizes the 6808's instructions and addressing modes. This 2-page Figure contains a wealth of information. For your convenience, this information is repeated on the Instruction Set Summary card provided with the course. You should keep this card handy. After a while, you will be able to write long, complex programs using only the card for reference.

The left-hand column of Figure 5-24 lists the names and mnemonics for each of the instructions. In many cases, a single name such as "add" is associated with more than one mnemonic. For example, ADDA is an add operation that involves accumulator A while ADDB is an add operation that involves accumulator B.

The center column gives important information about the addressing modes. Notice that the ADDA instruction can have any one of four addressing modes: immediate, direct, indexed, or extended. Three facts are given for each addressing mode. The hexadecimal opcode is given in the OP column. For example, the opcode for ADDA immediate is 8B while the opcode for ADDA direct is 9B.

The column labeled (~) tells the number of MPU cycles required to execute the instruction. This information is important because it allows us to determine exactly how long it will take to run a given program. As you will see later, an MPU cycle is equal to one cycle of the MPU clock. For example, if the clock frequency is 1 MHz, one MPU cycle will be one microsecond. With this clock rate, 2 microseconds are required to execute the ADDA immediate instruction while 5 microseconds are required for the ADDA indexed instruction.

The column labeled (#) indicates the number of bytes required by the instruction. ADDA immediate, ADDA direct, and ADDA indexed are two-byte instructions while ADDA extended is a three-byte instruction.

The next column to the right gives the shorthand notation for the Boolean or arithmetic operations performed. Finally, the right-hand column indicates how the condition code registers are affected by each instruction.

If you study the instruction set carefully, you will find that there are a few instructions that we have not yet discussed. These will be described in the next unit.

ACCUMULATOR AND MEMORY		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION						COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			INNER			(All register labels refer to contents)						5	4	3	2	1	0
		OP	~	#	OP	~	#	GP	~	#	OP	~	#	OP	~	#							H	I	N	Z	V	C
Add	ADDA	8B	2	2	9B	3	2	A8	5	2	BB	4	3				A + M → A											
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B											
Add Acmltrs	ABA													1B	2	1	A + B → A											
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A											
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B											
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A											
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B											
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M											
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M											
Clear	CLR							6F	7	2	7F	6	3				00 → M											
	CLRA													4F	2	1	00 → A											
	CLRB													5F	2	1	00 → B											
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A - M											
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B - M											
Compare Acmltrs	CBA													11	2	1	A - B											
Complement, 1's	COM							63	7	2	73	6	3				$\bar{M} \rightarrow M$											
	COMA													43	2	1	$\bar{A} \rightarrow A$											
	COMB													53	2	1	$\bar{B} \rightarrow B$											
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 - M → M											
	NEGA													40	2	1	00 - A → A											
	NEGB													50	2	1	00 - B → B											
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format											
Decrement	DEC							6A	7	2	7A	6	3				M - 1 → M											
	DECA													4A	2	1	A - 1 → A											
	DECB													5A	2	1	B - 1 → B											
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A											
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B											
Increment	INC							6C	7	2	7C	6	3				M + 1 → M											
	INCA													4C	2	1	A + 1 → A											
	INCB													5C	2	1	B + 1 → B											
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A											
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B											
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A											
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B											
Push Data	PSHA													36	4	1	A → M _{SP} , SP - 1 → SP											
	PSHB													37	4	1	B → M _{SP} , SP - 1 → SP											
Pull Data	PULA													32	4	1	SP + 1 → SP, M _{SP} → A											
	PULB													33	4	1	SP + 1 → SP, M _{SP} → B											
Rotate Left	ROL							69	7	2	79	6	3				M											
	ROLA													49	2	1	A											
	ROLB													59	2	1	B											
Rotate Right	ROR							66	7	2	76	6	3				M											
	RORA													46	2	1	A											
	RORB													56	2	1	B											
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M											
	ASLA													48	2	1	A											
	ASLB													58	2	1	B											
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M											
	ASRA													47	2	1	A											
	ASRB													57	2	1	B											
Shift Right, Logic.	LSR							64	7	2	74	6	3				M											
	LSRA													44	2	1	A											
	LSRB													54	2	1	B											
Store Acmltr	STAA							97	4	2	A7	6	2	B7	5	3	A → M											
	STAB							D7	4	2	E7	6	2	F7	5	3	B → M											
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A - M → A											
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B - M → B											
Subtract Acmltrs	SBA													10	2	1	A - B → A											
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A - M - C → A											
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B - M - C → B											
Transfer Acmltrs	TAB													16	2	1	A → B											
	TBA													17	2	1	B → A											
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M - 00											
	TSTA													4D	2	1	A - 00											
	TSTB													5D	2	1	B - 00											

Figure 5-24
The 6808 instruction set.

INDEX REGISTER AND STACK		IMMED			DIRECT			INDEX			EXTND			INNER			BOOLEAN/ARITHMETIC OPERATION							
POINTER OPERATIONS		MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#							
Compare Index Reg	CPX		8C	3	3	9C	4	2	AC	6	2	BC	5	3				(X _H /X _L) - (M/M + 1)	•	•	⑦	†	⑧	•
Decrement Index Reg	DEX														09	4	1	X - 1 → X	•	•	•	†	•	•
Decrement Stack Pntr	DES														34	4	1	SP - 1 → SP	•	•	•	•	•	•
Increment Index Reg	INX														08	4	1	X + 1 → X	•	•	•	†	•	•
Increment Stack Pntr	INS														31	4	1	SP + 1 → SP	•	•	•	•	•	•
Load Index Reg	LDX		CE	3	3	DE	4	2	EE	6	2	FE	5	3				M → X _H , (M + 1) → X _L	•	•	⑨	†	R	•
Load Stack Pntr	LDS		8E	3	3	9E	4	2	AE	6	2	BE	5	3				M → SP _H , (M + 1) → SP _L	•	•	⑨	†	R	•
Store Index Reg	STX					DF	5	2	EF	7	2	FF	6	3				X _H → M, X _L → (M + 1)	•	•	⑨	†	R	•
Store Stack Pntr	STS					9F	5	2	AF	7	2	BF	6	3				SP _H → M, SP _L → (M + 1)	•	•	⑨	†	R	•
Index Reg → Stack Pntr	TXS														35	4	1	X - 1 → SP	•	•	•	•	•	•
Stack Pntr → Index Reg	TSX														30	4	1	SP + 1 → X	•	•	•	•	•	•

JUMP AND BRANCH		RELATIVE			INDEX			EXTND			INNER			BRANCH TEST										
OPERATIONS		MNEMONIC	OP	~	=	OP	~	=	OP	~	=	OP	~	=										
Branch Always	BRA		20	4	2										None	•	•	•	•	•	•	•	•	•
Branch If Carry Clear	BCC		24	4	2										C = 0	•	•	•	•	•	•	•	•	•
Branch If Carry Set	BCS		25	4	2										C = 1	•	•	•	•	•	•	•	•	•
Branch If = Zero	BEQ		27	4	2										Z = 1	•	•	•	•	•	•	•	•	•
Branch If > Zero	BGE		2C	4	2										N ÷ V = 0	•	•	•	•	•	•	•	•	•
Branch If < Zero	BGT		2E	4	2										Z ÷ (N ÷ V) = 0	•	•	•	•	•	•	•	•	•
Branch If Higher	BHI		22	4	2										C + Z = 0	•	•	•	•	•	•	•	•	•
Branch If <= Zero	BLE		2F	4	2										Z ÷ (N ÷ V) = 1	•	•	•	•	•	•	•	•	•
Branch If Lower Or Same	BLS		23	4	2										C + Z = 1	•	•	•	•	•	•	•	•	•
Branch If <= Zero	BLT		2D	4	2										N ÷ V = 1	•	•	•	•	•	•	•	•	•
Branch If Minus	BMI		2B	4	2										N = 1	•	•	•	•	•	•	•	•	•
Branch If Not Equal Zero	BNE		26	4	2										Z = 0	•	•	•	•	•	•	•	•	•
Branch If Overflow Clear	BVC		28	4	2										V = 0	•	•	•	•	•	•	•	•	•
Branch If Overflow Set	BVS		29	4	2										V = 1	•	•	•	•	•	•	•	•	•
Branch If Plus	BPL		2A	4	2										N = 0	•	•	•	•	•	•	•	•	•
Branch To Subroutine	BSR		8D	8	2																			
Jump	JMP					6E	4	2	7E	3	3				} See Special Operations									
Jump To Subroutine	JSR					AD	8	2	BD	9	3													
No Operation	NOP														01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI														3B	10	1							
Return From Subroutine	RTS														39	5	1							
Software Interrupt	SWI														3F	12	1	} See special Operations	•	•	•	•	•	•
Wait for Interrupt	WAI														3E	9	1		•	•	•	•	•	•

CONDITIONS CODE REGISTER		INNER			BOOLEAN OPERATION						5 4 3 2 1 0													
OPERATIONS		MNEMONIC	OP	~	=	H I N Z V C						H I N Z V C												
Clear Carry	CLC		0C	2	1	0 • C	•	•	•	•	•	•	•	•	R	•	•	•	•	•	•	•	•	•
Clear Interrupt Mask	CLI		0E	2	1	0 • I	•	•	R	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Clear Overflow	CLV		0A	2	1	0 • V	•	•	•	•	•	•	•	•	R	•	•	•	•	•	•	•	•	•
Set Carry	SEC		0D	2	1	1 • C	•	•	•	•	•	•	•	•	•	S	•	•	•	•	•	•	•	•
Set Interrupt Mask	SEI		0F	2	1	1 • I	•	•	S	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Set Overflow	SEV		0B	2	1	1 • V	•	•	•	•	•	•	•	•	S	•	•	•	•	•	•	•	•	•
Accmltr A • CCR	TAP		06	2	1	A • CCR	⑫						•	•	•	•	•	•	•	•	•	•	•	
CCR • Accmltr A	TPA		07	2	1	CCR • A	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

LEGEND

OP Operation Code (Hexadecimal).

~ Number of MPU Cycles.

= Number of Program Bytes.

• Arithmetic Plus.

• Arithmetic Minus.

• Boolean AND.

M_{SP} Contents of memory location pointed to be Stack Pointer.

+ Boolean Inclusive OR.

• Boolean Exclusive OR.

M Complement of M.

• Transfer Into.

0 Bit = Zero.

00 Byte = Zero.

H Half carry from bit 3.

I Interrupt mask

N Negative (sign bit)

Z Zero (byte)

V Overflow, 2's complement

C Carry from bit 7

R Reset Always

S Set Always

• Test and set if true, cleared otherwise

• Not Affected

CCR Condrion Code Register

LS Least Significant

MS Most Significant

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of N ÷ C after shift has occurred.

⑦ (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?

⑧ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?

⑨ (Bit N) Test: Result less than zero? (Bit 15 = 1)

⑩ (All) Load Condition Code Register from Stack. (See Special Operations)

⑪ (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

⑫ (All) Set according to the contents of Accumulator A.

Figure 5-24

(continued)

Figure 5-24
(continued)

Self-Test Review

27. A disadvantage of direct addressing is that the operand must be in the first _____ bytes of memory.
28. The advantage of direct addressing is that only _____ bytes are required for each instruction.
29. Extended addressing can address _____ bytes of memory.
30. A disadvantage of extended addressing is that each instruction requires _____ bytes.
31. Can extended addressing be used to address an operand in the first 256_{10} bytes of memory?
32. The most powerful addressing mode available to the 6808 is called _____ addressing.
33. Indexed addressing requires _____ bytes for each instruction.
34. The second byte of an indexed addressing instruction is called the _____ address.
35. How is the address of the operand determined when indexed addressing is used?
36. Carefully examine the program shown in Figure 5-25. Determine what the program does and fill in the comments column. What number is loaded into the index register by the first instruction?

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	50	50	
0013	6F	→ CLR, X	
0014	00	00	
0015	08	INX	
0016	8C	CPX #	
0017	00	00	
0018	60	60	
0019	26	BNE	
001A	F8	F8	
001B	3E	WAI	

Figure 5-25
Program for Self-Test Review

37. What location is cleared by the CLR, X instruction?
38. What is the number in the index register after the INX instruction is executed for the first time?
39. The loop will be repeated until the number in the index register is _____.
40. What does this program do?
41. Refer to Figure 5-24. What is the hexadecimal opcode for the LDAB extended instruction?
42. How many MPU cycles are required by the INC, X instruction?
43. How many bytes in the LDS # instruction?

Answers

27. 256_{10} .
28. Two.
29. $65,536_{10}$.
30. Three.
31. Yes. Although direct addressing is normally used when the operand is in the first 256_{10} bytes of memory, extended addressing can be used also.
32. Indexed.
33. Two.
34. Offset.
35. The offset address is added to the contents of the index register.
36. 0050_{16} .

- 37. 0050_{16} .
- 38. 0051_{16} .
- 39. 0060_{16} .
- 40. The program clears memory locations 0050_{16} through $005F_{16}$.
- 41. $F6_{16}$.
- 42. Seven.
- 43. Three.

EXPERIMENTS

Perform Programming Experiments 7 and 8. You will find these experiments in Unit 7. After you finish these experiments, return to this unit and complete the Unit Examination.

UNIT EXAMINATION

1. Which of the following program segments will **not** clear both accumulators?
 - A. CLRA
CLRB
 - B. CLRA
TAB
 - C. CLRB
TBA
 - D. CLRA
ABA

2. Which of the following contains an operation that can **not** be performed directly on a byte in memory using a single instruction?
 - A. Increment, decrement, shift left arithmetically.
 - B. Clear, complement, compare.
 - C. Rotate left, negate, test for zero.
 - D. Shift right logically, rotate right, test for minus.

3. Which addressing mode is best suited for adding a list of numbers?
 - A. Direct.
 - B. Extended.
 - C. Indexed.
 - D. Relative.

4. Which of the following program segments will successfully swap the contents of accumulators A and B?

- | | |
|------------------------------------|------------------------------------|
| A. TAB
TBA | C. TAB
ABA |
| B. STAA
10
TBA
LDAB
10 | D. STAA
10
LDAB
10
TBA |

5. Which of the following program segments will cause a branch if the number in memory location 8310 is odd?

- | | |
|---------------------------------|------------------------------------------|
| A. ROR
83
10
BCS
07 | C. RORA
BCS
07 |
| B. ASL
83
10
BCS
07 | D. LDAA
83
10
ROLA
BCS
07 |

6. Examine the following program segment:

```

CLRA
→ INCA
  BNE
  FD
WAI

```

If an MPU cycle is 1 microsecond, how much time elapses from the time this segment starts running until the WAI instruction is fetched?

- A. Approximately 8 microseconds.
 B. Approximately 2050 microseconds.
 C. Approximately 1538 microseconds.
 D. Approximately 3 microseconds.

7. Which of the following instructions can be used to clear the Z flag?

- A. BEQ
- B. BNE
- C. NOP
- D. TAP

8. Which of the following instructions can be used to test the result of the subtraction of unsigned binary numbers?

- A. BGE.
- B. BGT.
- C. BCS.
- D. BLT

9. Examine the following program segment:

```
LDX #  
    00  
    50  
    → DEX  
    BNE  
    FD  
    WAI
```

How many times will the DEX instruction be executed?

- A. Once
- B. 50_{16} times.
- C. $65,536_{10}$.
- D. The number of times will depend on the contents of memory location 0050.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	4F	CLRA	Clear Accumulator A.
0011	7D	TST	Test
0012	00	00	the
0013	1E	1E	multiplier.
0014	27	BEQ	If it is zero branch to wait.
0015	07	07	
0016	7A	DEC	Otherwise decrement
0017	00	00	the
0018	1E	1E	multiplier.
0019	9B	ADDA	Add the
001A	1F	1F	multiplier to the product.
001B	20	BRA	Repeat the loop.
001C	F4	F4	
001D	3E	WAI	Wait.
001E	05	Multiplier	
001F	04	Multiplicand	

Figure 5-26

This program multiplies by repeated addition.

NOTE: Refer to the program shown in Figure 5-26 for questions 10 through 16.

10. What addressing mod does the TST instruction use?
 - A. Immediate
 - B. Direct.
 - C. Extended.
 - D. Indexed.
11. The BEQ instruction checks to see if the TST instruction set the:
 - A. Z flag
 - B. C flag.
 - C. H flag.
 - D. V flag.
12. The DEC instruction decrements the number in:
 - A. Accumulator A.
 - B. Memory location 001E.
 - C. Accumulator B.
 - D. The index register.

13. Which instruction is executed immediately after the BRA instruction?
- A. WAI.
 - B. BEQ.
 - C. CLRA.
 - D. TST.
14. With the values given for the multiplier and multiplicand, how many times will the main program loop be repeated?
- A. Four times.
 - B. Five times.
 - C. Twenty times.
 - D. Twice.
15. After the program has been executed, memory location 001E will contain:
- A. 05_{16} .
 - B. 04_{16} .
 - C. 20_{16} .
 - D. 00_{16} .
16. After the program has been executed, the product will appear in:
- A. Memory location 001E.
 - B. Memory location 001F.
 - C. Accumulator A.
 - D. Accumulator B.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	00	05	
0013	A6	LDAA, X	
0014	20	20	
0015	AB	ADDA, X	
0016	30	30	
0017	A7	STAA, X	
0018	40	40	
0019	08	INX	
001A	8C	CPX #	
001B	00	00	
001C	15	15	
001D	26	BNE	
001E	F4	F4	
001F	3E	WAI	

Figure 5-27

Program for Questions 17 through 20.

NOTE: Refer to Figure 5-27 for Questions 17 through 20. Analyze the program, determine what it does, and fill in appropriate comments.

17. On the first pass through the main program loop, the LDAA, X instruction takes its operand from memory location:
- A. 0005.
 - B. 0020.
 - C. 0025.
 - D. 0014.

18. On the first pass, the ADDA, X adds the contents of what memory location to accumulator A?
- A. 0005.
 - B. 0030.
 - C. 0035.
 - D. 0016.
19. On the second pass through the program loop, the contents of memory location:
- A. 0021 are added to the contents of 0031 and the result is stored in 0041.
 - B. 0026 are added to the contents of 0036 and the result is stored in 0046.
 - C. 0025 are added to the contents of 0035 and the result is stored in 0045.
 - D. 0020 are added to the contents of 0030 and the result is stored in 0040.
20. How many times is the main program loop repeated?
- A. 10_{16} times.
 - B. 05_{16} times.
 - C. 30_{16} times.
 - D. 15_{16} times.

EXAMINATION ANSWERS

1. D — The result is that both accumulators will contain whatever was in accumulator B when this program was executed.
2. B — The compare operation cannot be performed directly on a byte in memory.
3. C — Generally, indexed addressing is best suited for adding a list of numbers.
4. B — This program segment swaps the contents of accumulators A and B by using address 10 as a temporary storage location.
5. A — If the number is odd, its LSB will be 1. The ROR instruction shifts the LSB of location 8310 into the C bit. The BCS instruction then causes a branch if the C bit is set.
6. C — The loop composed of INCA and BNE will be repeated 256_{10} times. INCA requires 2 MPU cycles while BNE requires 4. If each MPU cycle is 1 microsecond, the time required is $256 \times 6 = 1536$ plus 2 microseconds required for the CLRA instruction. Thus, this segment takes about 1538 microseconds.
7. D — The TAP instructions can be used to set the condition of all flags at once.
8. C — The carry-borrow flag is set when a borrow occurs. Thus, the BCS instruction can be used to determine if the unsigned subtrahend was larger than the unsigned minuend.
9. B — The first instruction loads the index register with 0050_{16} . This number is repeatedly decremented until it reaches 0000. Thus DEX is executed 50_{16} times.

10. C — Extended.
11. A — The BEQ instruction tests the Z flag.
12. B — The DEC instruction decrements the number in memory location 001E.
13. D — The relative address (F4) directs the program back to the TST instruction.
14. B — The multiplier (05) is decremented on each pass until it reaches 00. Thus, the loop will be repeated five times.
15. D — The multiplier is reduced to 00 as the program is executed.
16. C — The product appears in accumulator A.
17. C — The offset address (20) is added to the number in the index register (0005) to form an operand address of 0025.
18. C — The contents of location 0035 are added to accumulator A on the first pass through the loop.
19. B — On the second pass, the contents of location 0026 are added to the contents of location 0036. The result is stored at 0046.
20. A — The index register starts at 0005 and is incremented to 0015. Therefore, the loop is repeated 10_{16} times.

Unit 6

**THE 6808 MICROPROCESSOR
PART 2**

CONTENTS

Introduction	6-3
Unit Objectives	6-4
Unit Activity Guide	6-5
Stack Operations	6-6
Subroutines	6-17
Input-Output (I/O) Operations	6-27
Interrupts	6-37
Experiments	6-51
Unit Examination	6-52
Examination Answers	6-55

INTRODUCTION

In the previous unit, you were introduced to the architecture and instruction set of the 6808 microprocessor. Much of the MPU's capabilities were discussed; however, three important areas were omitted. These include the microprocessor's stack operation, the use of subroutines, and the interrupt capability. These capabilities are discussed in detail in this unit. You are also introduced to input-output operations.

UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Explain the difference between a cascade stack and a memory stack.
2. Write simple programs that can store data in — and retrieve data from — the stack.
3. Write programs that use the stack and indexing to move a list from one place in memory to another.
4. Explain the operations performed by each of the following instructions: PULA, PULB, PSHA, PS HB, DES, INS, LDS, STS, TXS, and TSX.
5. Define stack, subroutine, nested subroutine, interrupt, interrupt vector, and interrupt masking.
6. Write programs that use subroutines and nested subroutines.
7. Explain the operations performed by each of the following instructions: JMP, JSR, BSR, and RTS.
8. Describe how the 6808 MPU performs input and output operations.
9. Draw flowcharts depicting the sequence of events that occur during reset, non-maskable interrupt, interrupt request, software interrupt, return from interrupt, and wait for interrupt.
10. Explain the operation performed by each of the following instructions: WAI, SWI, RTI, SEI, and CLI.

UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read the section on Stack Operations.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 1-10.	_____
<input type="checkbox"/> Read the section on Subroutines.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 11-20.	_____
<input type="checkbox"/> Read the section on Input-Output Operations.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 21-27.	_____
<input type="checkbox"/> Read the section on Interrupts.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 28-40.	_____
<input type="checkbox"/> Perform Programming Experiments 9 and 10.	_____
<input type="checkbox"/> Complete the Unit Examination.	_____
<input type="checkbox"/> Check the Examination Answers.	_____

STACK OPERATIONS

In computer jargon, a *stack* is a group of temporary storage locations in which data can be stored and later retrieved. In this regard, a *stack* is somewhat like memory. In fact, many microprocessors use a section of memory as a stack. The difference between a stack and other forms of memory is the method by which the data is accessed or addressed. The discussion will begin by considering a simple stack arrangement used in some microprocessors. Then the more sophisticated stack arrangement used by the 6808 MPU will be discussed.

Cascade Stack

Some microprocessors have a special group of registers (usually 8 or 16) called a *cascade stack*. Each register can hold one 8-bit byte of data. Because these registers are right on the MPU chip, they make excellent temporary storage locations. If we need to free the accumulator for some reason, we can store its contents in the stack. Later, if that piece of data is needed again, we can retrieve the data from the stack. Of course, we could also have freed the accumulator by storing the data in memory. What then is the advantage of the stack?

One advantage of the stack is the method by which it is accessed or addressed. Recall that when a byte is stored in memory, an address is required. That is to store the contents of the accumulator in memory a 2-byte or 3-byte instruction is required. Depending on the addressing mode, the last one or two bytes is the address. Later, if the byte is retrieved, another instruction is required that also has an address.

An advantage of the stack is that data can be stored into it or read from it with single-byte instructions. That is, the instructions used with the stack do not require an address. Therefore, they are single-byte instructions.

Figure 6-1 shows an 8-register stack similar to that found in some microprocessors. This is called a cascade stack because of the method by which data is loaded and retrieved. All data transfers are between the top of the stack and the accumulator. That is, the accumulator communicates *only* with the top location on the stack. Data is transferred to the stack by a special instruction called PUSH.

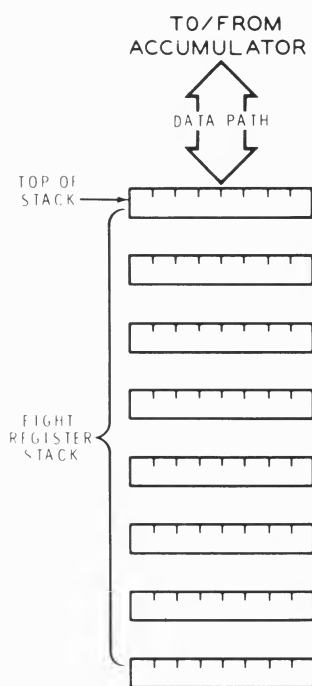


Figure 6-1.
A cascade stack.

The PUSH Instruction. Figure 6-2 illustrates how the PUSH instruction places data in the stack. The number 01_{16} is in the accumulator and we wish to temporarily store it. While we could store the number in memory, this would require a 2-byte or a 3-byte instruction. So instead, we use the PUSH instruction to place this number in the stack. Notice that the number is placed in the top location of the stack as shown in Figure 6-2A. The number remains there until we retrieve it or until we push another byte into the stack.

Figure 6-2B shows what happens if, at some time later, we push another byte into the stack. Notice that the accumulator now contains 03_{16} . If the PUSH instruction is executed, the contents of the accumulator are pushed into the top of the stack. To make room for this new number, the original number 01_{16} is pushed deeper into the stack.

Figures 6-2C and 6-2D show two more numbers being pushed into the stack at later points in the program. Notice that new data is always pushed into the top of the stack. To make room for the new data, the old data is pushed deeper into the stack. For this reason, this arrangement is often called a *push-down* or *cascade stack*. The name *cascade stack* comes from the characteristic cascading of data down through the stack as each new byte is pushed in at the top.

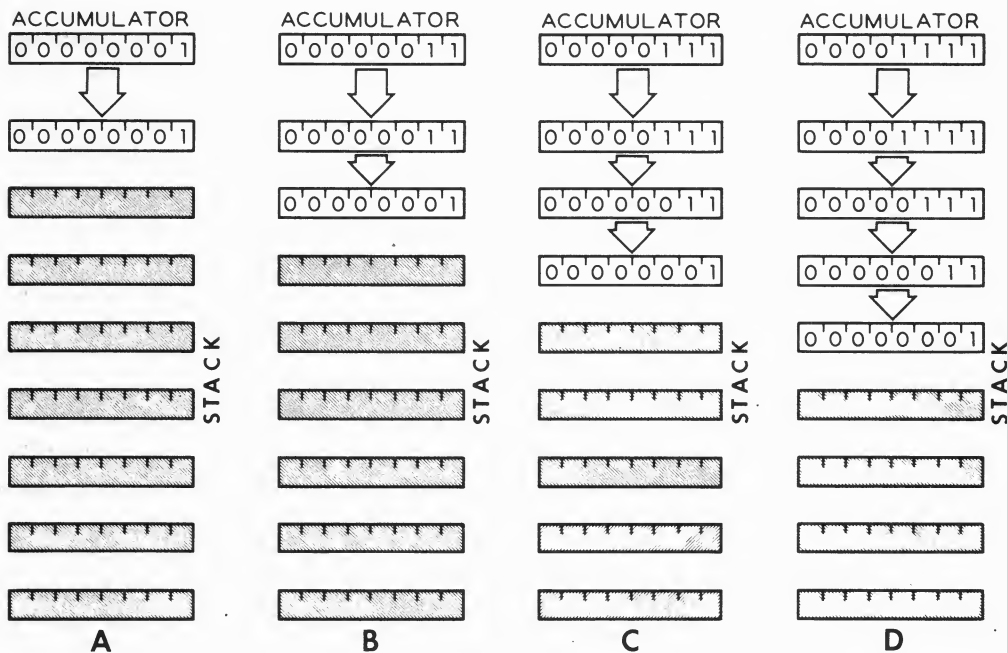


Figure 6-2.
Pushing data into the stack.

The PULL Instruction. The MPU retrieves data from the stack by using the PULL instruction. In some microprocessors, this is referred to as a POP instruction.

Figure 6-3 illustrates how data can be pulled (or popped) from the stack. Figure 6-3A shows the stack as it appeared after the last push operation. Notice that it contains four bytes of data. The last byte of data that was entered is at the top of the stack.

The PULL instruction retrieves the byte that is at the top of the stack. As this byte is removed from the stack, all other bytes move up, filling in the space left by that byte. Figure 6-3B illustrates how $0F_{16}$ is pulled from the stack. Notice that 07_{16} is now at the top of the stack.

Figures 6-3C and 6-3D show how the next two bytes can be pulled from the stack. In each case, the remaining bytes move up in the stack, filling in the register vacated by the removed byte.

If you compare Figures 6-2 and 6-3, you will notice that the data must be pulled from the stack in the reverse order. That is, the last byte pushed into the stack is the first byte that is pulled from the stack. Another name for this arrangement is a last-in/first-out (LIFO) stack.

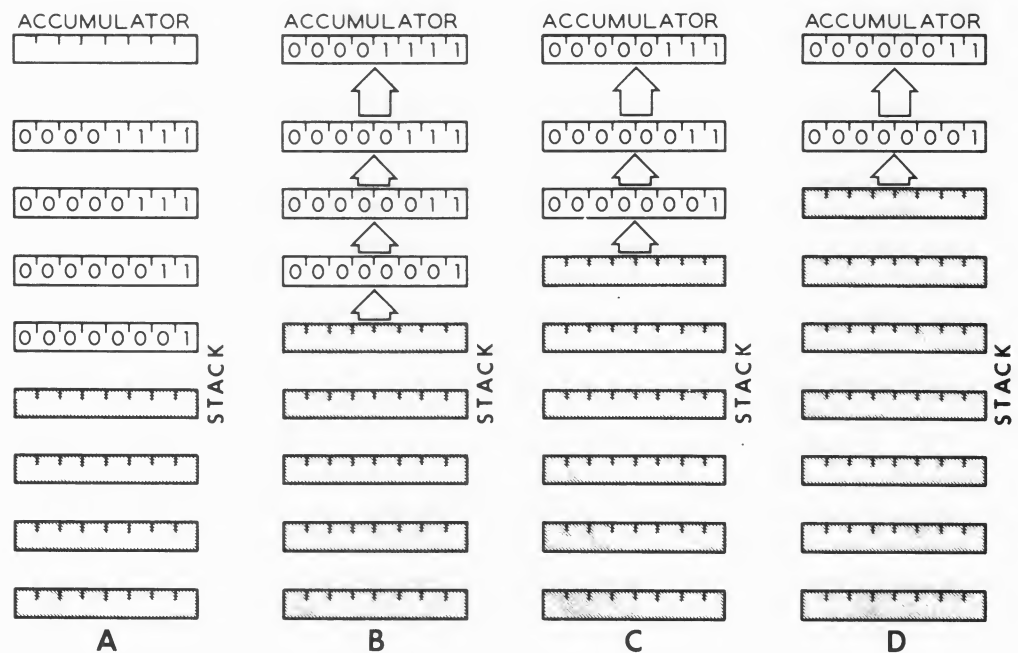


Figure 6-3.
Pulling data from the stack.

Memory Stack

While a cascade stack is valuable, it does have some limitations. For one thing, the number of registers is generally quite limited, with eight being typical. If more than eight pieces of data are pushed into the stack, the “older” bytes are pushed out the bottom and are lost. Also, the readout of the stack is destructive. When a byte is pulled from the stack, it no longer exists in the stack. This is fundamentally different from reading a byte from memory.

Because of these limitations the 6808 MPU does not use a cascade stack. Instead, a section of RAM can be set aside by the programmer to act as a stack. This has several advantages. First, the stack can be any length that the programmer requires. Second, the programmer can set up more than one stack if he likes. Third he can address the data in the stack using any of the instructions that address memory.

Stack Pointer. Recall that the 6808 MPU has a 16-bit register called the stack pointer. In a memory-type stack, the stack pointer defines the memory location that acts as the top of the stack.

The cascade stack considered earlier generally does not require a stack pointer. The top of the stack is determined by hardware. During push and pull operations, the data bytes actually move from one register to another. That is, the top of the stack remains stationary and the data moves up or down in relation to the stack.

In the memory stack, data cannot be easily transferred from one location to the next. Therefore, instead of moving data up and down in relation to the stack, it is much easier to move the top of the stack in relation to the data.

Generally, when the microprocessor-based system is being planned, a section of RAM is reserved for the stack. This should be a section of RAM that is not being used for any other purpose.

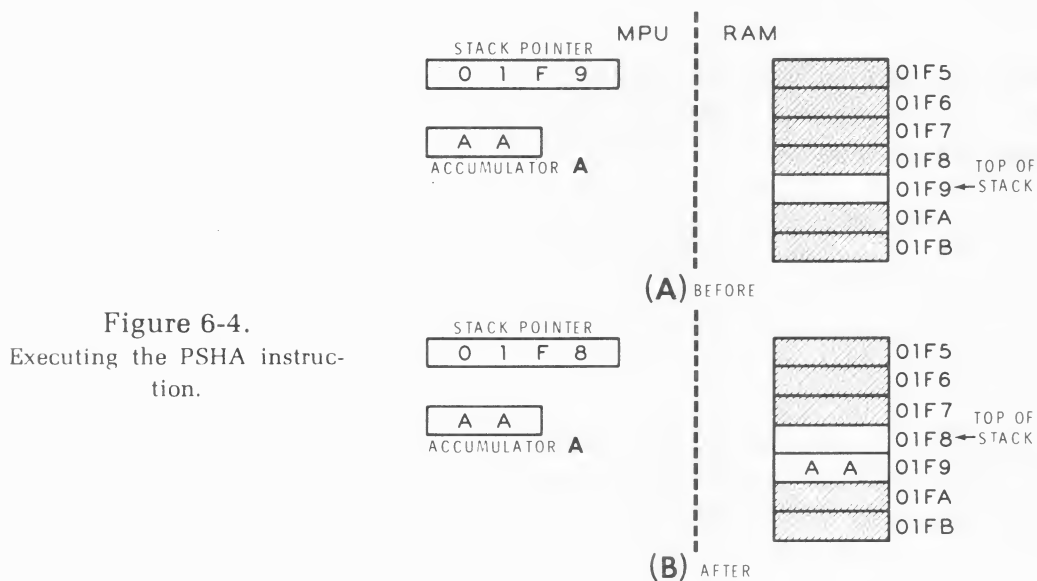
Once this is done, the stack can be set up by a program. The top of the stack is established by loading an address into the stack pointer. For example, suppose we wish to establish address $01F9_{16}$ as the top of the stack. The following instruction could be used:

```
LDS#  
01  
F9
```

This loads the address $01F9_{16}$ into the stack pointer and establishes that address as the top of the stack. However, as you will see, the top of the stack moves each time data is pushed into — or pulled from — the stack.

The PUSH Instructions. The 6808 MPU has two push instructions, PSHA and PSHB. These single-byte instructions push the contents of their respective accumulator onto the stack.

Figure 6-4 shows the effects of the PSHA instruction. Before the instruction is executed, the stack pointer contains the address $01F9_{16}$ as a result of a previous LDS instruction. Accumulator A contains a data byte (AA_{16}). If the PSHA instruction is now executed, the contents of accumulator A are pushed into memory location $01F9_{16}$. Then, the stack pointer is automatically decremented to $01F8_{16}$. This automatically moves the top of the stack as shown.



If you look at your Instruction Set Summary card, you will see that the operation is described as follows:

$$A \rightarrow M_{SP}, SP - 1 \rightarrow SP$$

This means that the contents of the A accumulator are transferred to the memory location specified by the stack pointer. Also, the contents of the stack pointer are replaced by the previous contents of the stack pointer minus one. In other words, after the accumulator-to-stack transfer takes place, the stack pointer is decremented by one.

To reinforce the idea, assume that at some later point in the program, the MPU executes a PSHB instruction. This is illustrated in Figure 6-5. Before PSHB is executed, the B accumulator contains BB_{16} and the stack pointer is still pointing to $01F8_{16}$. When PSHB is executed, the contents of accumulator B are pushed onto the stack and the stack pointer is decremented to $01F7_{16}$.

The PULL Instructions. Data bytes are removed from the stack with the pull instruction. The 6808 MPU has two pull instructions. PULA allows the MPU to pull data from the stack into the A accumulator. PULB performs a similar operation except the data byte goes into accumulator B. In each case, data is pulled from the top of the stack. Thus, the data byte available to the MPU is the last byte that was placed in the stack.

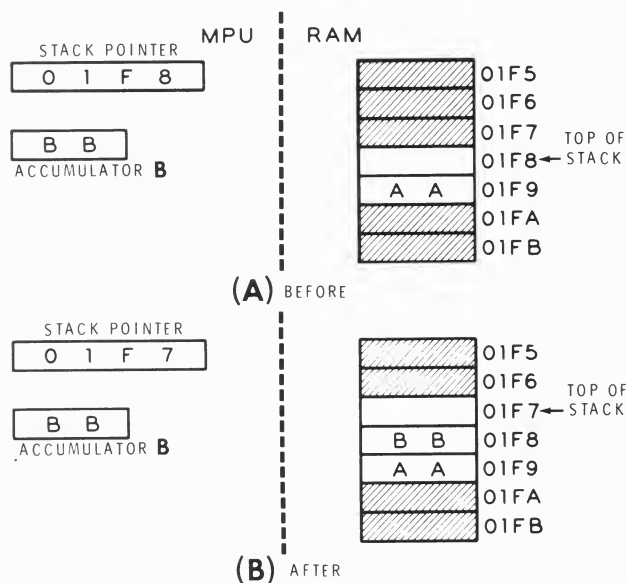


Figure 6-5.
Executing the PSHB instruction.

For example, Figure 6-6A shows the stack as we left it after the last push instruction. Figure 6-6B shows what happens if the PULA instruction is executed. First, the stack pointer is automatically incremented by one to $01F8_{16}$. Then the contents of the memory location designated by the stack pointer are transferred to accumulator A. Thus, BB_{16} goes into accumulator A. Notice that the stack pointer is incremented *before* the byte is pulled from the stack.

To be certain you have the idea, consider what happens if the PULB instruction is now executed. Figure 6-6C shows that the stack pointer is automatically incremented to $01F9_{16}$. The contents of that location are then pulled into accumulator B. This operation is described on your Instruction Set Summary card as:

$$SP + 1 \rightarrow SP, M_{SP} \rightarrow B.$$

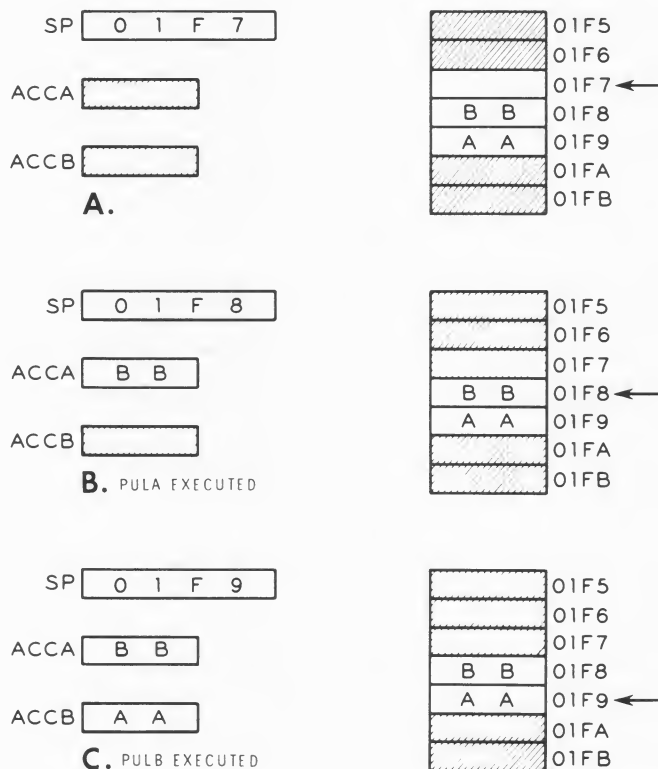


Figure 6-6.
Executing PULL instructions.

Using the Stack. Figure 6-7 summarizes all of the instructions that directly affect stack operations. The push and pull instructions were introduced in this unit while the other instructions were discussed briefly in the previous unit. Find these instructions on your Instruction Set Summary card. The push and pull instructions are listed with the Accumulator and Memory Operations. Those instructions that affect the stack pointer are listed under Index Register and Stack Pointer Operations.

		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)
STACK AND STACK POINTER		IMMED			DIRECT			INDEX			EXTND			INHER			
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	=	OP	~	#	OP	~	#	
Push Data	PSHA													36	4	1	A → M _{SP} , SP - 1 → SP B → M _{SP} , SP - 1 → SP SP + 1 → SP, M _{SP} → A SP + 1 → SP, M _{SP} → B SP - 1 → SP SP + 1 → SP M → SP _H , (M + 1) → SP _L SP _H → M, SP _L → (M + 1) X - 1 → SP SP + 1 → X
	PSHB													37	4	1	
Pull Data	PULA													32	4	1	
	PULB													33	4	1	
Decrement Stack Pntr	DES													34	4	1	
Increment Stack Pntr	INS													31	4	1	
Load stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				
Indx Reg → Stack Pntr	TXS													35	4	1	
Stack Pntr → Indx Reg	TSX													30	4	1	

Following are some examples of how the stack can be used. First consider a trivial example. Using only stack operations, swap the contents of accumulators A and B. Assuming the stack pointer has already been set up, the program segment might look like this:

```
PSHA
PSHB
PULA
PULB
```

Assume that accumulator A initially contains AA_{16} and that accumulator B contains BB_{16} . The first instruction pushes AA_{16} onto the stack. Next BB_{16} is pushed onto the stack. The third instruction pulls BB_{16} from the top of the stack and places it in accumulator A. Finally, the last instruction pulls AA_{16} from the stack and places it in accumulator B. As you can see, the contents of the two accumulators are reversed. The following routine accomplishes the same thing with one less instruction:

```
PSHA
TBA
PULB
```

Figure 6-7.
Stack and stack pointer instructions.

Now look at a more complex example. Assume that you wish to transfer 16_{10} bytes of data from one place in memory to another. As you saw in the previous unit, this type of problem is a good candidate for indexing. However, indexing alone becomes cumbersome if the two lists are over FF_{16} memory locations apart. The reason for this is that the offset address can only extend FF_{16} locations above the address in the index register.

In this example, assume you wish to move the data in memory locations 0010_{16} through $001F_{16}$ to locations $01F0_{16}$ to $01FF_{16}$. While this could be accomplished using indexing alone, the program becomes unnecessarily complicated. Two separate indexes must be maintained; one for loading data from 0010_{16} through $001F_{16}$, the other for storing data in $01F0_{16}$ through $01FF_{16}$. A simpler approach is to use indexing for one operation and the stack capability for the other operation. That is, we could load data from the lower list using indexing and store it in the upper list using the stack capability.

A program that does this is shown in Figure 6-8. The first instruction loads the stack pointer with address $01FF_{16}$. This is the address of the last entry in the new list that will be formed. Recall that the new list is to be written in locations $01F0_{16}$ through $01FF_{16}$. Once location $01FF_{16}$ is established as the top of the stack, we can enter data into the new list simply by pushing data onto the stack. Because the stack pointer is decremented with each push operation, we must push the last entry in the list onto the stack first.

Figure 6-8.
Moving a list of data using both
indexing and stack operations.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0020	8E	LDS#	Load the stack pointer immediately with the
0021	01	01	address of the last entry in the
0022	FF	FF	new list.
0023	CE	LDX#	Load the index register immediately with the
0024	00	00	address of the last entry in the
0025	1F	1F	original list.
0026	A6	LDAA, X	Load accumulator A indexed from
0027	00	00	the original list.
0028	36	PSHA	Push the contents of accumulator A into the new list.
0029	09	DEX	Decrement the index register.
002A	8C	CPX#	Compare the contents of the index register
002B	00	00	with one less than the address of the
002C	0F	0F	first entry in the original list.
002D	26	BNE	If no match occurs, branch back
002E	F7	F7	this far.
002F	3E	WAI	Otherwise, wait.

The second instruction loads the index register with the address of the last entry in the original list. This is necessary for the reason pointed out above.

Next, the A accumulator is loaded using indexed addressing. Since the offset address is 00_{16} , the accumulator is loaded with the contents of $001F_{16}$. That is, the last entry in the original list is loaded into accumulator A.

The PSHA instruction then pushes the contents of accumulator A onto the stack. Thus, the last entry in the original list is transferred to location $01FF_{16}$. In the process, the stack pointer is automatically decremented to $01FE_{16}$.

The index register is decremented to $001E_{16}$ by the next instruction. Then, the CPX instruction compares the index register with $000F_{16}$ to see if all entries in the list have been moved. If no match occurs, the MPU branches back and picks up the next entry in the list. The loop is repeated over and over again until the entire list has been moved to its new location.

Other uses of the stack will be revealed later. However, even if the stack did nothing more than has already been explained, it would be a very useful capability to have. But as you will see, the MPU uses the stack in several other ways that makes this capability even more important.

Self-Test Review

1. What is a stack?
2. What is a cascade stack?
3. What is a memory stack?
4. Which type of stack does the 6808 MPU use?
5. What is the name of the instruction that stores data in the stack?
6. What type of instruction is used to retrieve data from the stack?
7. What is the purpose of the stack pointer?
8. The PUSH instruction transfers data from one of the accumulators to _____.
9. The PULB instruction transfers data from the top of the stack to _____.
10. Refer to Figure 6-8. How can we change this program so that the new list is placed in addresses 0220_{16} through $022F$?

Answers

1. A stack is a group of registers or a section of memory that is used as a last-in, first-out memory.
2. A cascade stack is a group of hardware registers (usually 16 or less) that is used as a last-in, first-out memory.
3. A memory stack uses a section of RAM as a last-in, first-out memory.
4. A memory stack
5. PUSH
6. PULL
7. The stack pointer indicates the address of the top of the stack.
8. The top of the stack.
9. Accumulator B.
10. By changing the first instruction to: $\text{LDS\# } 022\text{F}_{16}$.

SUBROUTINES

A subroutine is a group of instructions that performs some limited but frequently required task. A given subroutine may be used many times during the execution of the main program. In many cases, the easiest way to write a program is to break the overall job down into many simple operations, each of which can be performed by a subroutine.

Because subroutines are used so frequently, most microprocessors have special capabilities that allow them to handle subroutines efficiently. In this section, these capabilities will be examined. The discussion will start with the instructions associated with subroutines.

The 6808 MPU has three instructions that are used to handle subroutines. They are:

Jump to Subroutine (JSR)
Branch to Subroutine (BSR)
Return from Subroutine (RTS)

Each of these will be discussed in this section. One other instruction that has not yet been mentioned will also be discussed. It is the Jump (JMP) instruction. While not used exclusively with subroutines, the JMP instruction makes an excellent introduction to the Jump to Subroutine (JSR) instruction. Therefore, the Jump (JMP) instruction will be discussed first.

Jump (JMP) Instruction

This instruction allows the MPU to jump from one point in a program to another. In this respect, it is somewhat like the Branch Always (BRA) instruction that was discussed earlier. The difference is the method of addressing used. Recall that the BRA instruction used relative addressing. This has the advantage that only a 2-byte instruction is required. Its disadvantage is that the branch must be within the range of -128 bytes to $+127$ bytes of the program count.

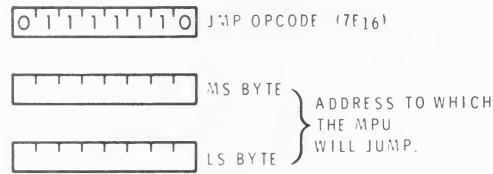


Figure 6-9.
Format of the JMP instruction
using extended addressing.

The JMP instruction can use either the indexed or the extended addressing mode. It does not use relative addressing. When using extended addressing, the format of the JMP instruction is as shown in Figure 6-9. Three bytes are required; the opcode followed by the 2-byte address to which the MPU is to jump. Since a 16-bit address is given, the jump may be to any point in the 65,536₁₀ byte memory range. This address is loaded into the program counter so that the next opcode is fetched from that address. The previous contents of the program counter are lost. Thus, the MPU starts executing instructions from a new point in memory.

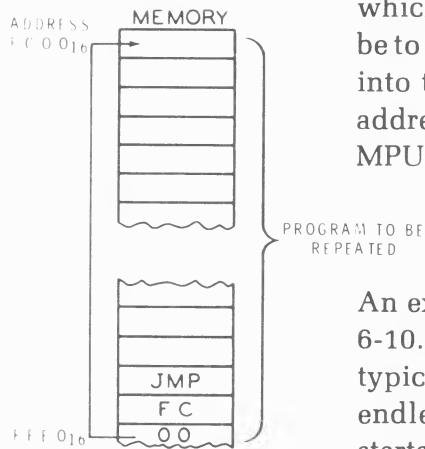


Figure 6-10.
Using the JMP instruction to
repeat a program.

An example of how the JMP instruction can be used is shown in Figure 6-10. Here, a long program is to be repeated over and over again. This is typical of applications such as controllers that repeat the same operations endlessly. The program is contained in the upper 1k bytes of memory. It starts at location FC00₁₆ and ends at FFE0₁₆. Notice that the last instruction is JMP FC00₁₆. This sends the program back to its beginning so that the loop is repeated endlessly.

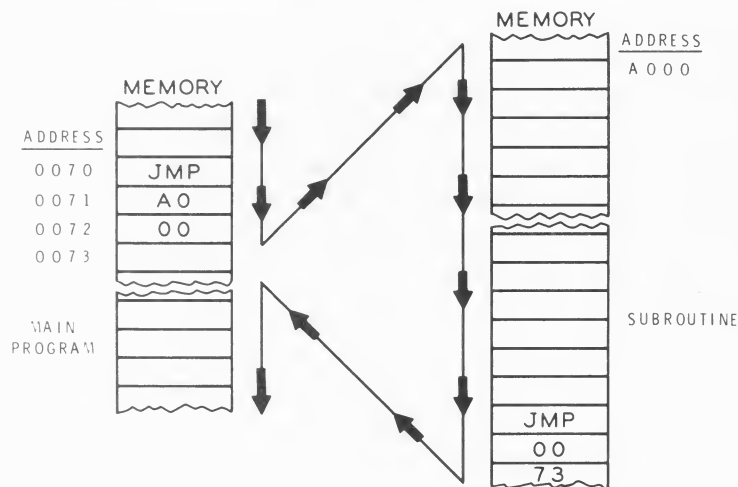


Figure 6-11.
Using the JMP instruction to
call a subroutine.

Another possible use of the JMP instruction is shown in Figure 6-11. Here, the main program is in the lower memory locations shown on the left. The main program requires a subroutine that is up at address A000 (shown on the right). The JMP instruction at address 0070 sends the MPU off to the subroutine as shown. The last instruction in the subroutine is another JMP instruction that sends the MPU back to the main program.

Jumping to a subroutine is often referred to as *calling* a subroutine. While we can call a subroutine using the JMP instruction, this approach has a distinct problem. What happens if the main program wants to call the same subroutine more than once? That is, suppose a situation like that shown in Figure 6-12 is required. Here, the main program (on the left) wishes to call the subroutine (on the right) at two separate points. Jumping to the subroutine is no problem. We can do that as many times as we like, using the instruction JMP A000. The problem is: how do we get back from the subroutine to the main program? The first time through the subroutine, the MPU should return to address 0073. The second time through, the MPU should return to address 0093.

A programmer could get around this problem by changing the last instruction in the subroutine before each call or by constructing a table of return addresses, etc. However, most microprocessors have some instructions that solve this problem for us. The following section will discuss the 6808 MPU's solution to this problem.

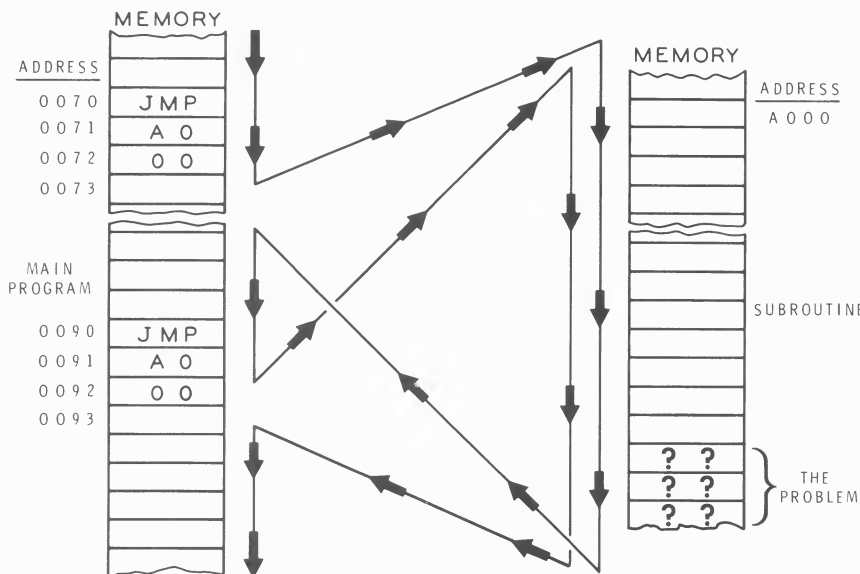


Figure 6-12.
The JMP instruction cannot
handle situations like this one.

When the first JSR instruction is executed, the subroutine address $A000_{16}$ is placed in the program counter. However, just prior to this, the program counter was incremented to the address of the next instruction in sequence. That is, the program counter was advanced to 0073_{16} while the contents of address 0072_{16} were being retrieved. This count (0073_{16}) is automatically pushed onto the stack. By saving the old program count, the MPU can tell where to return after the subroutine is finished. As soon as the old program count is tucked away safely in the stack, the subroutine address $A000_{16}$ is placed in the program counter. Thus, the MPU fetches the next instruction from address $A000_{16}$.

Notice that the last instruction in the subroutine is an RTS instruction. When the MPU encounters this single-byte instruction, it will jump back to the point where it left off in the main program. It does this by pulling the old program count (0073_{16}) from the stack and placing it in the program counter. Consequently, the next instruction will be fetched from address 0073_{16} . As you can see, this returns the MPU to the correct point in the main program.

Notice that the programmer does not specify a return address at the end of the subroutine. The return address is automatically pulled from the stack. This allows us to call the subroutine repeatedly from several different points in the main program.

Figure 6-13 shows that the subroutine is called again by the JSR $A000$ instruction in location 0090_{16} . As this instruction and address are decoded, the program count is incremented to 0093_{16} . This program count is pushed onto the stack. Then $A000_{16}$ is placed in the program counter. Thus, the MPU jumps off to the subroutine. When the subroutine is finished, the RTS instruction causes the old program count to be pulled from the stack into the program counter. This causes the MPU to jump back to address 0093_{16} which contains the next instruction in the main program.

Nested Subroutines

Figure 6-14 shows a situation in which the main program calls subroutine A. In turn, subroutine A calls subroutine B. In this situation, subroutine B is called a *nested subroutine*. That is, a nested subroutine is a program segment that is called by another subroutine. If control is to be eventually returned to the main program, two program counts must be saved. Figure 6-14 shows how the two program counts are saved in the stack.

At the start of the main program, the stack pointer is loaded with the address of the area in memory that has been set aside to act as the stack. If no stack instructions have been executed when the main program arrives at the first JSR instruction, the stack pointer will still be pointing to where it was originally set. The contents of the stack are of no interest until this point.

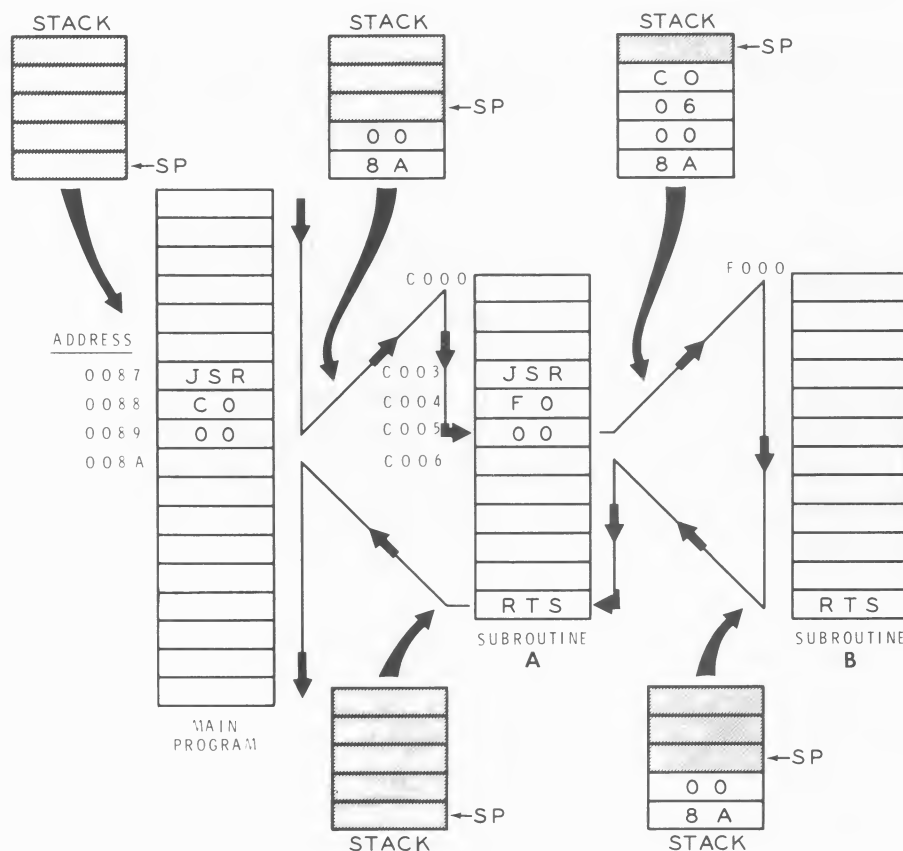


Figure 6-14.
Handling nested subroutines.

When the main program reaches the JSR instruction, the program count is advanced to the address of the next instruction in sequence ($008A_{16}$). When the JSR instruction is executed, this address ($008A_{16}$) is pushed onto the stack as shown. The low order byte goes in first, followed by the high order byte. In the process, the stack pointer is decremented twice. Finally, the new address ($C000_{16}$) is placed in the program counter. This causes the MPU to jump off to subroutine A which starts at $C000_{16}$.

Notice that halfway through subroutine A, subroutine B is called. Consequently, the return address in subroutine A ($C006_{16}$) must be saved. That is, when the program reaches the JSR instruction in subroutine A, the return address ($C006_{16}$) is pushed onto the stack as shown. Notice that there are now two return addresses in the stack. The starting address of subroutine B ($F000_{16}$) is then placed in the program counter and the MPU jumps off to this subroutine.

Subroutine B has no nested subroutines of its own, so the program flow is through the subroutine as shown. The last instruction in subroutine B is the RTS instruction. At this point, the MPU pulls the return address ($C006_{16}$) from the top of the stack and places it in the program counter. This causes the MPU to jump back to the instruction at address $C006_{16}$ in subroutine A.

The remainder of subroutine A is then executed down to the RTS instruction. This instruction causes the MPU to pull the next address ($008A_{16}$) from the stack and place it in the program counter. Notice that this sends the MPU back to the main program.

For simplicity, a single level of subroutine nesting is shown in this example. However, in practice, many levels of nesting may be used. For example, subroutine B could call subroutine C; etc. Any level of nesting can be used as long as enough memory is set aside for the stack. Remember, each return address requires two bytes in the stack.

Branch to Subroutine (BSR) Instruction

Quite often, the subroutine we wish to call is within the -128_{10} to $+127_{10}$ byte range of the relative address. When it is, we can save one byte by using the Branch to Subroutine (BSR) instruction. The execution of BSR is identical to that of JSR except that relative addressing is used. The old program count is saved in the stack before the branch occurs. Thus, the RTS instruction at the end of the subroutine will cause the old program count to be restored.

Summary of Subroutine Instructions

Figure 6-15 shows the four instructions discussed in this section. Notice that the BSR instruction uses relative addressing. The JMP and JSR instructions can use either indexed or extended addressing. The RTS instruction uses inherent addressing since its address is pulled from the top of the stack.

Find these instructions on your Instruction Set Summary card. The operations performed by these instructions are illustrated under "Special Operations" on the back of the card. Also, Appendix A of this course gives a concise description of the operations performed by each of these instructions.

JUMP AND BRANCH OPERATIONS		MNEMONIC		RELATIVE		INDEX		EXTND		INHER	
				OP	#	OP	#	OP	#	OP	#
Branch To Subroutine	BSR	8D	8	2							
Jump	JMP				6E	4	2	7E	3	3	
Jump To Subroutine	JSR				AD	8	2	BD	9	3	
Return From Subroutine	RTS										39 5 1

Figure 6-15.
Subroutine and jump instructions.

Self-Test Review

11. What is a subroutine?
12. What addressing modes can the JMP instruction use?
13. How does the JMP instruction differ from the BRA instruction?
14. How does the execution of the JSR instruction differ from that of the JMP instruction?
15. Why is the program count saved when the JSR or BSR instructions are executed?
16. Where is the program count saved?
17. How is the stack pointer affected by the JSR instruction?
18. Generally, the last instruction in the subroutine will be a _____ instruction.
19. What is a nested subroutine?
20. How is the stack pointer affected by the RTS instruction?

Answers

11. A subroutine is a group of instructions that performs some specific, limited task that is used more than once by the main program.
12. Indexed and extended.
13. Since the BRA instruction uses relative addressing, it can branch only in a -128_{10} to $+127_{10}$ byte range. The JMP instruction uses indexed or extended addressing. Therefore, it can jump to any point in memory.
14. When the JSR instruction is executed, the program count is saved in the stack.
15. The program count is saved so that when the subroutine is finished, the MPU can return to the point it left off.
16. The program count is pushed into the top two locations of the stack.
17. The stack pointer is automatically decremented twice as the program count is pushed onto the stack.
18. Return from Subroutine (RTS).
19. When subroutine A calls subroutine B, subroutine B is said to be nested.
20. The stack pointer is automatically incremented twice as the old program count is pulled from the stack.

INPUT — OUTPUT (I/O) OPERATIONS

A full explanation of input-output (I/O) operations will be given in the next units, but a brief introduction to I/O is necessary at this point. In this section, you will learn what is involved in sending data to — or taking data from — the MPU.

To be useful, a microprocessor system must accept data from the outside world, process it in some way, and present results to the outside world. The input device may be nothing more than a group of switches while the output device can be as simple as a bank of indicator lamps. On the other hand, a single microprocessor might handle several teletypewriters, printers, papertape machines, etc. The point is that the I/O requirements can vary greatly from one application to the next. This section will be concerned with the simplest form of I/O operations.

In the short history of microprocessors, two distinctly different methods have been developed for handling I/O operations. In some microprocessors, I/O operations are handled by I/O instructions. These microprocessors generally have one *input* instruction and one *output* instruction. When the *input* instruction is executed, a byte is transferred from the selected I/O device to a register (usually one of the accumulators) in the MPU. The I/O device is selected by sending out a device selection byte on the address bus. By using an 8-bit byte for device selection, the MPU can specify up to 256_{10} different I/O devices. Of course, no microprocessor system uses that many devices, but the capability is there. The *output* instruction causes a data transfer from the accumulator to the selected I/O device. While this method of handling I/O operations is used in many microprocessors, the 6808 MPU uses a different technique.

The other method for handling I/O operations is to treat all I/O transfers as memory transfers. This is the method used by the 6808 MPU and many other microprocessors. In fact, even those microprocessors that have I/O instructions can ignore those instructions and handle I/O operations as memory transfers.

The 6808 MPU has no I/O instructions. An I/O device is assigned an address and is treated as a memory location. For example, assume that an input keyboard has been assigned an address of 8000_{16} . We can input data into accumulator A by using the instruction:

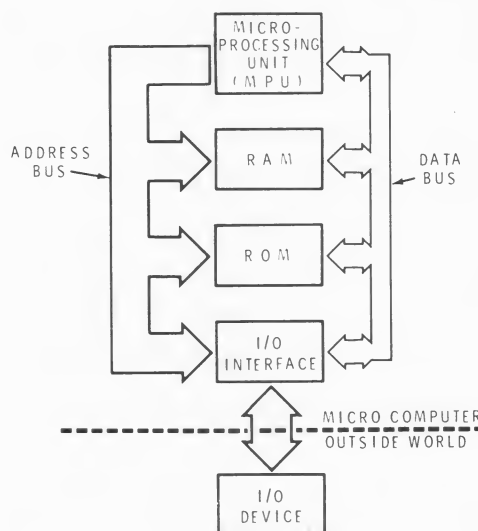
LDAA 8000_{16}

By the same token, an output display may have been assigned the address 9000_{16} . In this case, we can output from accumulator B by using the instruction:

STAB 9000_{16} .

As you can see, the I/O device is treated as a memory location. The system block diagram shown in Figure 6-16 shows how an I/O device is connected to the microcomputer. Notice that both the data bus and the address bus connect to the I/O interface. As you will see in the next unit, the interface can consist of an address decoder, an output or input latch, and buffers or drivers.

Figure 6-16.
Adding I/O to the microcomputer.



The address decoder monitors the address bus and enables the interface circuitry whenever the proper address is detected. This prevents the I/O interface from interfering when data is being transferred between memory and the MPU.

The I/O interface will generally have an output latch if it is to be used for an output operation. The reason for this is that the data from the MPU will appear on the data lines for only an instant (usually less than one microsecond). By storing the output data in a latch, the I/O device is given a much longer period of time to examine and respond to the data.

Buffers or drivers are also included in the I/O interface. As you will see later, these are frequently necessary when several different circuits are sharing the same bus.

Output Operations

Figure 6-17 shows a simplified output circuit. Here, the output device is a bank of eight light emitting diodes (LEDs). Enough detail is shown to illustrate how an output operation can be performed. The address decoder monitors the address bus, looking for the address 9000_{16} . It also monitors some of the control lines that connect to the MPU. One of those lines is called a read-write line. It goes to its low state when a write (output) operation is initiated by the MPU. The other control lines will be discussed in the next unit.

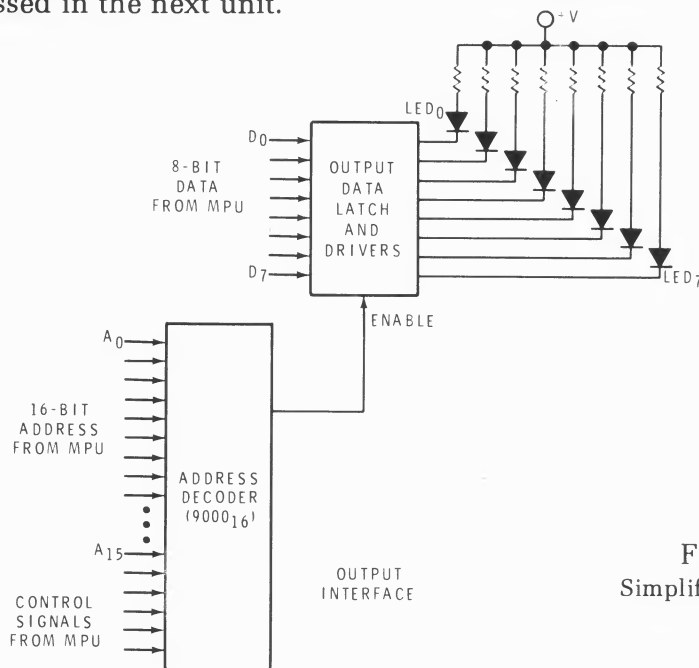


Figure 6-17.
Simplified output circuit.

Notice that the output of the address decoder is used to enable the output data latch and drivers. When these are enabled, the byte on the data lines is stored in the latch. The data bits stored in the latch cause the appropriate LEDs to light up. By outputting appropriate bit patterns, the MPU can cause different binary numbers to be displayed.

Notice that the address decoder (and therefore the display) is given the address 9000_{16} . We can output data to the display in several different ways. For example, we can load the appropriate pattern to be displayed into accumulator A. Then by executing a “store accumulator A” extended instruction, we can transfer the contents of the accumulator to the display. The instruction would be: STAA 9000_{16} . Or, we could output data from accumulator B by using the instruction: STAB 9000_{16} .

In either case, the address 9000_{16} goes out on the address bus for a brief interval of time. The address decoder recognizes this address. At the same time, the control lines indicate that an output operation is called for. In particular, the read-write line goes low. This causes the address decoder to enable the output data latch for an instant. Simultaneously, the 8-bit data byte appears on the data bus. The output latch stores the data byte. The data appears at the input of the latch for less than a microsecond (typically). However, once the data is stored, it appears at the output of the latch until new data is written in. Thus, the output data will be displayed until the next byte of data is outputted by the MPU.

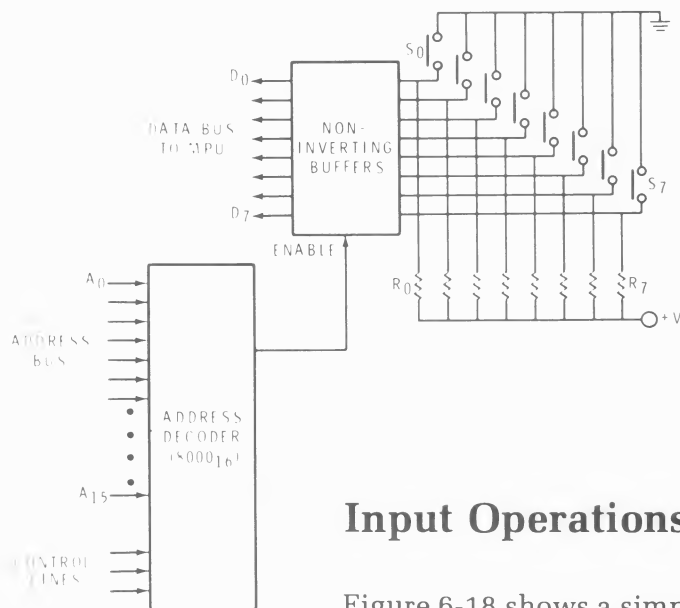


Figure 6-18.
Simplified input circuit.

Input Operations

Figure 6-18 shows a simplified input circuit. Here, the input device is a bank of eight switches. When a switch is open, its respective input line to the buffer is held high by the pull-up resistors. However, when a switch is closed, its respective input line is pulled low because the switch connects it to ground.

In this simple circuit, no latch is required between the switches and the data bus. However, a buffer is used so that the switch bank can be effectively disconnected from the data bus when the switches are not being addressed.

As with the output circuit, an address decoder monitors the address and control lines. Notice that the assigned address is 8000_{16} . To input data from the switch bank to accumulator A, we use the instruction: `LDAA 8000_{16}` . Or, we could input the data to accumulator B by using the instruction: `LDAB 8000_{16}` .

In either case, the address 8000_{16} is placed on the address line. The address decoder recognizes this address and enables the buffer. For a brief interval (typically less than one microsecond), the lines of the data bus assume the same state as the lines on the right side of the buffer. If no switch is depressed, all data lines will be high and all 1's (FF_{16}) will be loaded into the accumulator. However, if one of the switches (S_0 , for example) is depressed, its respective data line (D_0) will be low. In this case, the number read into the accumulator will be FE_{16} . By examining the byte that is read in, the MPU can determine which switch is depressed.

Input — Output Programming

You now know enough about simple input/output circuits to perform some I/O operations. Refer to Figures 6-17 and 6-18. For the first example, assume that you would like one of the LEDs to light when the corresponding switch is pushed. That is, LED_0 should light when S_0 is pushed; LED_1 should light when S_1 is pushed, etc.

If you refer to Figure 6-17, you will see that an LED is caused to light by placing a 0 in the proper bit in the latch. For example, a 0 in bit 0 will cause LED_0 to be forward biased. Thus, the diode will conduct and emit light. Notice that a 1 at bit 0 will not allow the diode to conduct and emit light. Consequently, a 0 turns the LED on and a 1 turns it off.

Refer to Figure 6-18, and you will find that, when one of the switches is closed, its corresponding line goes to 0. If the switch is not closed, its corresponding line is at 1.

If we load data into one of the accumulators from address 8000_{16} and then store the data at address 9000_{16} , the switches will appear to control the LED's. The program could look like this:

```
LDAA  
80  
00  
STAA  
90  
00  
BRA  
F8
```

If S_0 , and only S_0 , is closed when the LDAA 8000 instruction is executed, 11111110_2 will be loaded into accumulator A. The next instruction stores this data byte in the output latch. This causes LED₀, and only LED₀, to light. The BRA instruction holds the MPU in a tight loop. Try a few examples and verify that each time a switch is closed, the corresponding LED will light. If the switches are set to some 8-bit binary number, the LED's will display that 8-bit number.

Now, suppose we change our mind and decide that the LEDs should display the one's complement of the binary number set on the switches. We do not have to touch the hardware. Instead, we just change the program. The new program might look like this:

```
      → LDAA
        80
        00
        COMA
        STAA
        90
        00
        BRA
        F7
```

Notice that we have simply inserted the one's complement instruction between the input and output operations.

As another example, suppose we wish to display a number that is four times greater than the number set on the switches. Our program could be changed to this:

```
      → LDAA
        80
        00
        ASLA
        ASLA
        STAA
        90
        00
        BRA
        F6
```

Once again, no hardware change is needed. We simply insert two ASLA instructions between the input and output operations.

Although these examples are very simple, they illustrate the flexibility of this I/O arrangement. Data is pulled from the input device as if it were being pulled from memory. Once in the MPU, the data byte can be modified in any way we like. The data can then be transferred to the output device as if it were being stored in memory. While the data is in the MPU, it can be modified in any number of ways. The input byte can be shifted left or right. It can be added to — or subtracted from — another number. It can be ANDed or ORed with another byte. The possibilities are endless and yet none of these involve a hardware change. All data manipulations can be accomplished by the program.

Program Control of I/O Operations

In the preceding examples, all I/O transfers are controlled by the program and the program alone. The program is in a tight loop that inputs data from the switches, modifies the data (if required), and outputs the data to the displays.

When this arrangement is used, the MPU never knows if the data at the input has changed. It simply reads in the data a number of times each second. By the same token, the MPU outputs the data over and over again. This system works well for simple I/O operations. However, as the I/O requirements become more sophisticated, this technique becomes cumbersome.

The program must be in a loop if it is to repeatedly check for inputs and refresh the output. As the number of data manipulations increase, the loop becomes longer and the MPU must check the inputs less frequently. When several I/O devices are used, it must check each input and refresh each output repeatedly. If the loop becomes too long, the MPU may miss a momentary switch closure. This may be acceptable in some applications but in many others it may be intolerable. Obviously then, a more sophisticated method of handling I/O operations must be available to the microcomputer.

Interrupt Control of I/O Operations

A more effective way of handling I/O operations involves a concept called *interrupts*. Interrupts are a means by which an I/O device can notify the MPU that it is ready to send input data or to accept output data. Generally, when an interrupt occurs, the MPU suspends its current operation and takes care of the interrupt. That is, it might read in or write out a byte of data. After it has taken care of the interrupt, the MPU returns to its original task and takes up where it left off.

An analogy may help you to visualize an interrupt operation. Compare the MPU to the president of a corporation who is writing a report. The interrupt can be compared to a telephone call. The president's main task is the report. However, if the telephone rings (an interrupt), she finishes writing the present word or sentence then answers the phone call. After she has attended to the phone call, she returns to the report and takes up where she left off. In this analogy, the ringing of the telephone notifies the president of the interrupt request.

This analogy shows the difficulty of the program controlled I/O technique discussed earlier. If we remove the interrupt request (the ringing of the phone), we are left with an almost comical situation. The president writes a few words of the report. She then picks up the phone to see if anyone is on the other end. If not, she hangs up the phone, writes a few more words, and checks the phone again. Clearly, this technique wastes an important resource — the president's time.

This simple analogy shows the importance of an interrupt capability. Without it, a great deal of the MPU's time can be wasted doing routine operations. The next section will examine the interrupt capabilities of the 6808 MPU.

Self-Test Review

21. What are the two methods by which microprocessors handle I/O operations?
22. Which method does the 6808 MPU use?
23. Which instruction can be used for transferring data from an I/O device to accumulator A?
24. Which instruction can be used for transferring data from accumulator B to an I/O device.
25. Write a program segment that will: read in data from the switch bank shown in Figure 6-18; double the number; and display the result on the LED bank shown in Figure 6-17.
26. What is meant by program control of an I/O operation?
27. What is meant by interrupt control of an I/O operation?

Answers

- 21. Some microprocessors have input-output instructions; others treat I/O as memory.
- 22. The 6808 MPU treats I/O as memory.
- 23. LDAA
- 24. STAB
- 25. One solution is:

LDAA
80
00
ASLA
STAA
90
00

- 26. Using this method, the program regularly reads in or writes out data. All I/O operations are controlled by the program.
- 27. Using this method, the I/O device itself signals the MPU that it is ready to transmit or receive data. The I/O operations are controlled largely by the I/O device itself.

INTERRUPTS

Interrupts were introduced in the previous section in connection with I/O operations. While I/O operations use part of the interrupt capability of the MPU, interrupts are also used in other ways. The 6808 MPU has four different types of interrupts:

Reset
Non-Maskable Interrupt (NMI)
Interrupt Request (IRQ)
Software Interrupt (SWI)

This section will examine each of these interrupts in detail.

Reset

In a typical application, the microcomputer has a control or monitor program in a read-only-memory (ROM). Also, a random access read-write memory (RAM) is used for holding input data, intermediate answers, output data, etc. As we have seen, the 6808 MPU has the capability of addressing up to $65,536_{10}$ memory locations. Most microprocessor applications do not require this much memory. In many applications, the control program requires less than ten percent of the possible locations. The RAM probably uses less than two percent. Generally, the monitor program is placed at the high memory addresses. The RAM is usually given the low memory addresses so that the direct addressing mode can be used. The I/O devices are given intermediate addresses. Thus, the memory addresses may be allocated as shown in Figure 6-19.

Notice that the control or monitor program is placed in a ROM at the very top of memory. In this example, a 1024_{10} byte ROM is used. The addresses of the ROM are $FC00_{16}$ through $FFFF_{16}$. A small RAM is placed at the low end of memory. Addresses 0000_{16} through $01FF_{16}$ are used. Notice that all other addresses are unused except for two. The input device is assigned address 8000_{16} , while the output device is assigned address 9000_{16} .

The monitor program stored in the ROM, controls all the activities of the MPU. At all times, the entire system is being run by this program. In this example, when the microprocessor is initially turned on, it should start executing instructions at address $FC00_{16}$. Also, we should be able to restart the program at this address at any time. In order to accomplish this, the 6808 MPU has a built-in reset capability.

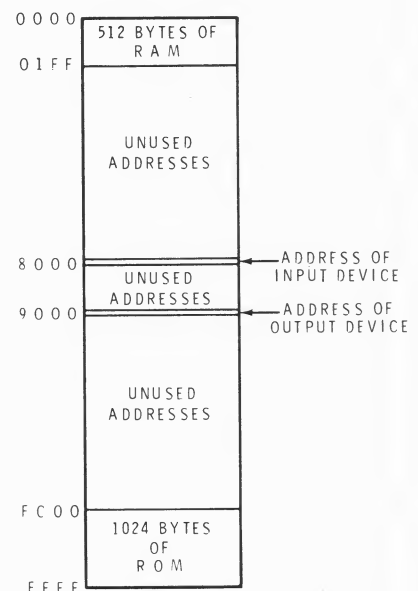


Figure 6-19.
Memory allocations in a typical microcomputer system.

The 6808 MPU has a signal line or control pin that is called $\overline{\text{Reset}}$. This pin or line is connected to a reset switch of some kind. If this line goes low for a prescribed period of time (to be explained later) and then swings high, the MPU will initiate a *reset interrupt sequence*. The main purpose of the reset interrupt sequence is to load the address of the first instruction to be executed into the program counter. This would be easy to accomplish if, in every application, the starting address were the same. However, the starting address differs from one application to the next. Therefore, a convenient means is provided to allow the designer to specify any starting address that he likes.

In any 6808 based microprocessor system, the upper eight bytes of ROM are reserved for *interrupt vectors*. An interrupt vector is simply an address that is loaded into the program counter when an interrupt occurs. Figure 6-20 shows how these eight reserved memory bytes are allocated. Notice that addresses FFFE_{16} and FFFF_{16} contain the reset vector. That is, these two memory locations contain the address of the first instruction that is to be executed when the microcomputer is initially started. In our example, the first instruction in the monitor program is at address FC00_{16} . Consequently, this is our reset vector. Location FFFE_{16} must contain the high byte of the address (FC_{16}) and FFFF_{16} must contain the low byte of the address (00_{16}).

Remember locations FFFE_{16} and FFFF_{16} are in the read-only-memory. Therefore, the designer must provide the proper reset vector at the time he is writing the monitor program.

Address	
F F F 8	Interrupt Request Vector (high order address)
F F F 9	Interrupt Request Vector (low order address)
F F F A	Software Interrupt Vector (high order address)
F F F B	Software Interrupt Vector (low order address)
F F F C	Non-Maskable-Interrupt Vector (high order address)
F F F D	Non-Maskable-Interrupt Vector (low order address)
F F F E	Reset Vector (high order address)
F F F F	Reset Vector (low order address)

Figure 6-20.
Interrupt vector assignments.

Figure 6-21 shows the sequence of events that occurs when the MPU is reset. First, the interrupt (I) mask bit is set. You will recall that the I flag is one of the condition code registers. As you will see later, if this flag is set, it prevents one of the other interrupts from occurring. Thus, the MPU sets the interrupt mask bit so that the reset sequence will not be interrupted by a request for interrupt by one of the I/O devices.

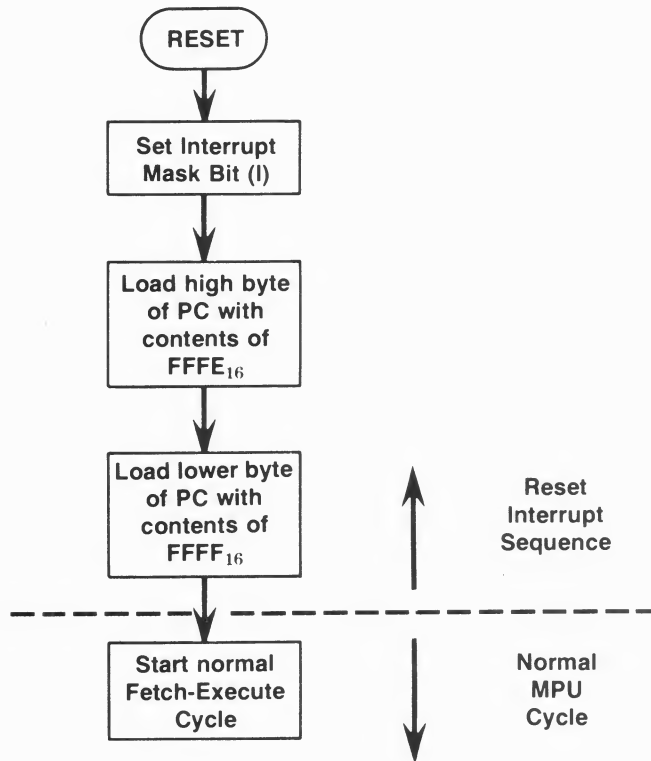


Figure 6-21.
Reset interrupt sequence.

Second, the contents of location $FFFE_{16}$ are loaded into the high byte of the program counter. This is done by sending the address $FFFE_{16}$ out on the address bus. The memory location is read out and its contents are placed on the data bus. The MPU picks up this byte and places it in the upper eight bits of the program counter. In our example, the byte in location $FFFE_{16}$ is FC_{16} .

Next, the contents of location $FFFF_{16}$ are loaded into the lower eight bits of the program counter. This is done by setting the address bus to $FFFF_{16}$. Thus, the contents of the highest memory location are placed on the data bus. In our example, this byte is 00_{16} . At this point, the program counter contains the address of the first instruction which is $FC00_{16}$.

The reset sequence is then terminated by switching the MPU to its normal fetch-execute machine cycle. Thus, the instruction at address $FC00_{16}$ is fetched and executed. From this point on, all MPU activities are controlled by the program.

The microprocessor system will have a reset switch somewhere in the system. This will allow the operator to restart the system if the system locks up or runs away for some reason. In addition, some systems will have an automatic reset feature that will allow the system to reset itself after a power failure. In both cases, the reset capability of the MPU is used.

This reset capability can be considered an interrupt, since the MPU leaves whatever it is doing and jumps off to the start of the monitor program. In most cases, the monitor program would start with a short subroutine that initializes the system. It would do things like set up the stack pointer, initialize displays, etc.

Non-Maskable Interrupts

The 6808 has two other types of hardware interrupts. One of these interrupts is maskable; the other is not. A maskable interrupt is one that the MPU can ignore under certain conditions. Whereas, a non-maskable interrupt cannot be ignored. To illustrate the difference, recall the corporation president analogy.

The president's report writing can be interrupted by the telephone. However, by telling her secretary to hold all calls, she has effectively masked one source of interruptions. In this analogy it is impractical to mask all interrupts. For example, it could be counterproductive to mask the fire alarm.

Somewhat the same situation can exist in a microprocessor controlled system. Some interrupts can be ignored for a few seconds while the MPU is performing a more important task. This type of interrupt can be masked. Others must not be ignored at all. These cannot be masked. Of course, it is up to the designer to decide which interrupts can be masked and which cannot. The 6808 MPU has provisions for handling both types. How the MPU handles the non-maskable type will be discussed first.

The 6808 MPU has a control line called the non-maskable interrupt (NMI) line. A high-to-low transition on this line forces the MPU to initiate a non-maskable interrupt sequence. The purpose of this sequence is to provide an orderly means by which the MPU can jump off to a service routine that will take care of the interrupt.

This becomes somewhat involved because the MPU must be able to go back to its main program after the interrupt service routine is finished. It must be able to pick up exactly where it left off. Furthermore, all registers must hold exactly the same data and addresses that they held when the

interrupt occurred. In other words, when an interrupt occurs, the program count must be saved so that the MPU can later return to this point in the program. Also, the contents of the accumulators, index register, and even the condition code registers must be saved so that the MPU can be restored to the exact condition that existed at the instant the interrupt occurred.

The 6808 MPU accomplishes this by pushing all the pertinent data onto the stack. Then, after the interrupt has been serviced, the MPU returns to its previous status by pulling the data from the stack.

The non-maskable interrupt sequence is shown in Figure 6-22. A non-maskable interrupt is initiated when the $\overline{\text{NMI}}$ line goes from its high state to its low state. The MPU finishes the execution of the current instruction. However, before another instruction is fetched, the MPU pushes the contents of its registers onto the stack. Recall that the stack pointer always points to the top of the stack. For this example, assume that the stack pointer was set by an earlier instruction to address 0068_{16} .

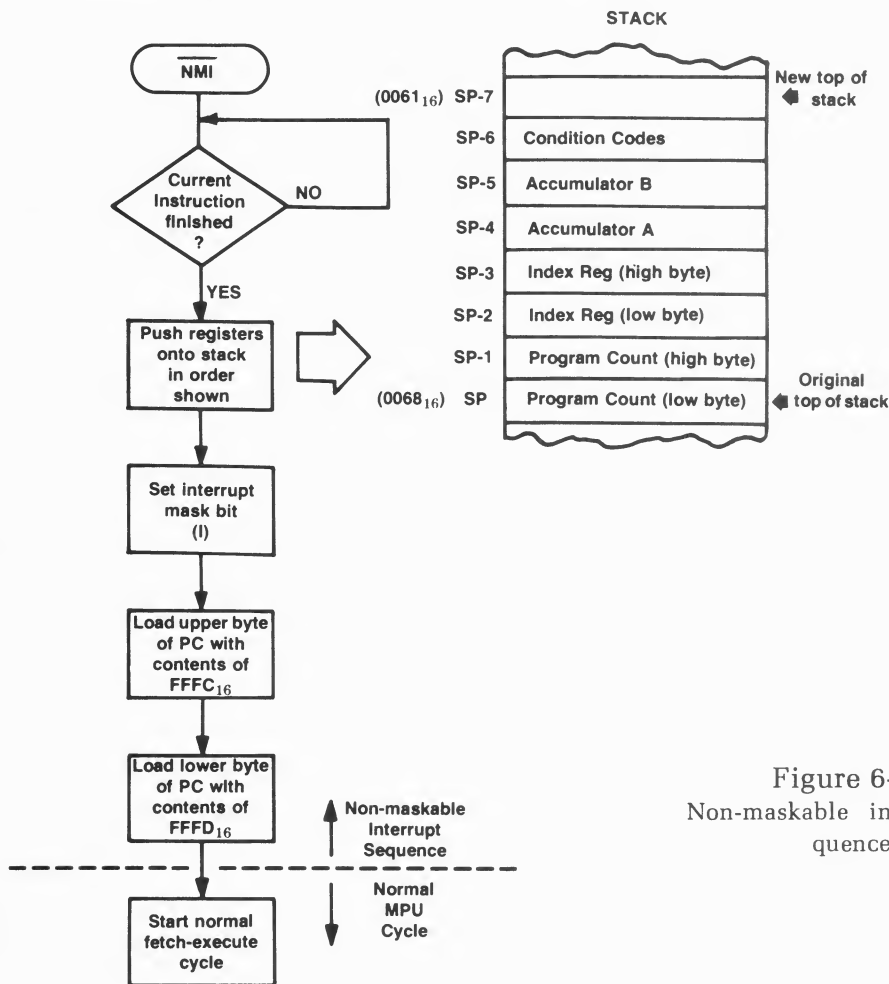


Figure 6-22.
Non-maskable interrupt sequence.

The MPU pushes the lower eight bits of the program counter into memory location 0068_{16} . Then it decrements the stack pointer so that the upper eight bits of the program counter are pushed into address 0067_{16} . Next, the contents of the index register are pushed into addresses 0066_{16} and 0065_{16} . The contents of accumulators A and B and the condition codes are also pushed in as shown. When all this has been done, the stack pointer will have been decremented seven times to 0061_{16} .

Return to the flow chart and notice that the next step is to set the interrupt mask bit. This allows the MPU to ignore any interrupt requests that occur while the non-maskable interrupt is being serviced.

At this point, the MPU is ready to jump to the interrupt service routine. But, what is the address of this routine? Recall the interrupt vector chart that was shown earlier in Figure 6-20. The non-maskable interrupt vector is at addresses $FFFC_{16}$ and $FFFD_{16}$. Thus, the upper byte of the program counter is loaded from $FFFC_{16}$ while the lower byte is loaded from $FFFD_{16}$. This directs the MPU to the first instruction in the non-maskable interrupt service routine. From this point on, the MPU returns to its normal fetch-execute cycle until the service routine is finished.

The sequence of events shown in Figure 6-22 happen automatically when a non-maskable interrupt sequence is initiated. The \overline{NMI} line gives external hardware a method of forcing a jump-to-subroutine to occur. In this case, the subroutine is a short program that performs some action to take care of the interrupt.

Return From Interrupt (RTI) Instruction

The non-maskable interrupt is used when some situation exists that cannot be ignored. You can probably visualize applications that would require such a capability. For example, assume that a microprocessor is being used in a numerically controlled drill press. The non-maskable interrupt could be used in conjunction with limit switches to prevent drilling holes in the work surface. Or, it could be used to shut down the machine if someone's hand got too close.

The purpose of the service routine is to direct the operation of the computer to take care of the interrupt. Typically, it would first determine which external device initiated the interrupt. Then it would determine the nature of the interrupt. Finally, it would take whatever action was necessary to take care of the interrupt. In many cases, the interrupt is of a routine nature and can be easily serviced. In these situations, the MPU

should return to the main program and take up where it left off. There is an instruction that allows the MPU to do this. It is called the “Return-From-Interrupt” (RTI) instruction. Look on your Instruction Set Summary card, and you will see that this is a one-byte instruction whose opcode is $3B_{16}$.

Figure 6-23 shows how the RTI instruction is used. The main program is shown on the left, while the interrupt service routine is shown on the right. Assume that the interrupt signal occurs while the LDAB# instruction is being executed. The MPU finishes that one instruction and pushes all pertinent data onto the stack. It then jumps to an address determined by the \overline{NMI} vector in address FFFC and FFFD. The contents of these two locations determine the starting address of the \overline{NMI} service routine. Notice that the last instruction in the service routine is the return-from-interrupt instruction. This instruction returns program control to the point in the main program that the MPU left when the interrupt occurred.

This can be done because the previous status of the MPU was preserved in the stack. The RTI instruction causes the accumulators, the index registers, the condition code register, and the program counter to be loaded from the stack. Thus, the same information that went into the stack when the interrupt occurred comes out of the stack when the RTI instruction is executed. This allows the MPU to return to the main program and take up where it left off.

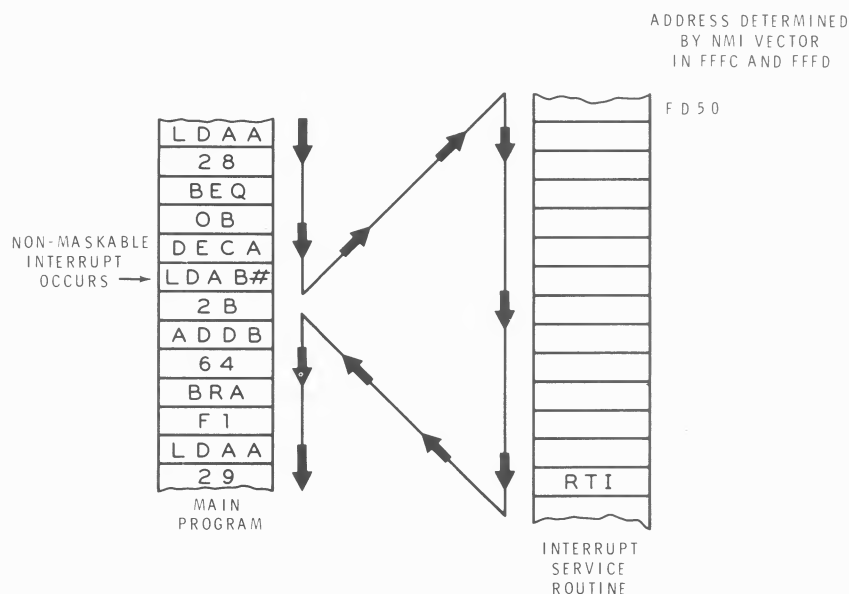


Figure 6-23.

The RTI instruction returns control to the main program after the interrupt has been serviced.

Interrupt Request (IRQ)

The interrupt request is very similar to the non-maskable interrupt. The main difference between the two is that the interrupt request is maskable.

The 6808 MPU has a control line called the interrupt request ($\overline{\text{IRQ}}$) line. When this line is low, an interrupt sequence is requested. However, the MPU may or may not initiate the interrupt sequence depending on the state of the interrupt mask (I) bit in the condition code register. If the I bit is set, the MPU ignores the interrupt request. If the I bit is not set, the MPU initiates the interrupt sequence. This procedure is very similar to the $\overline{\text{NMI}}$ procedure discussed earlier. Figure 6-24 shows the interrupt procedure.

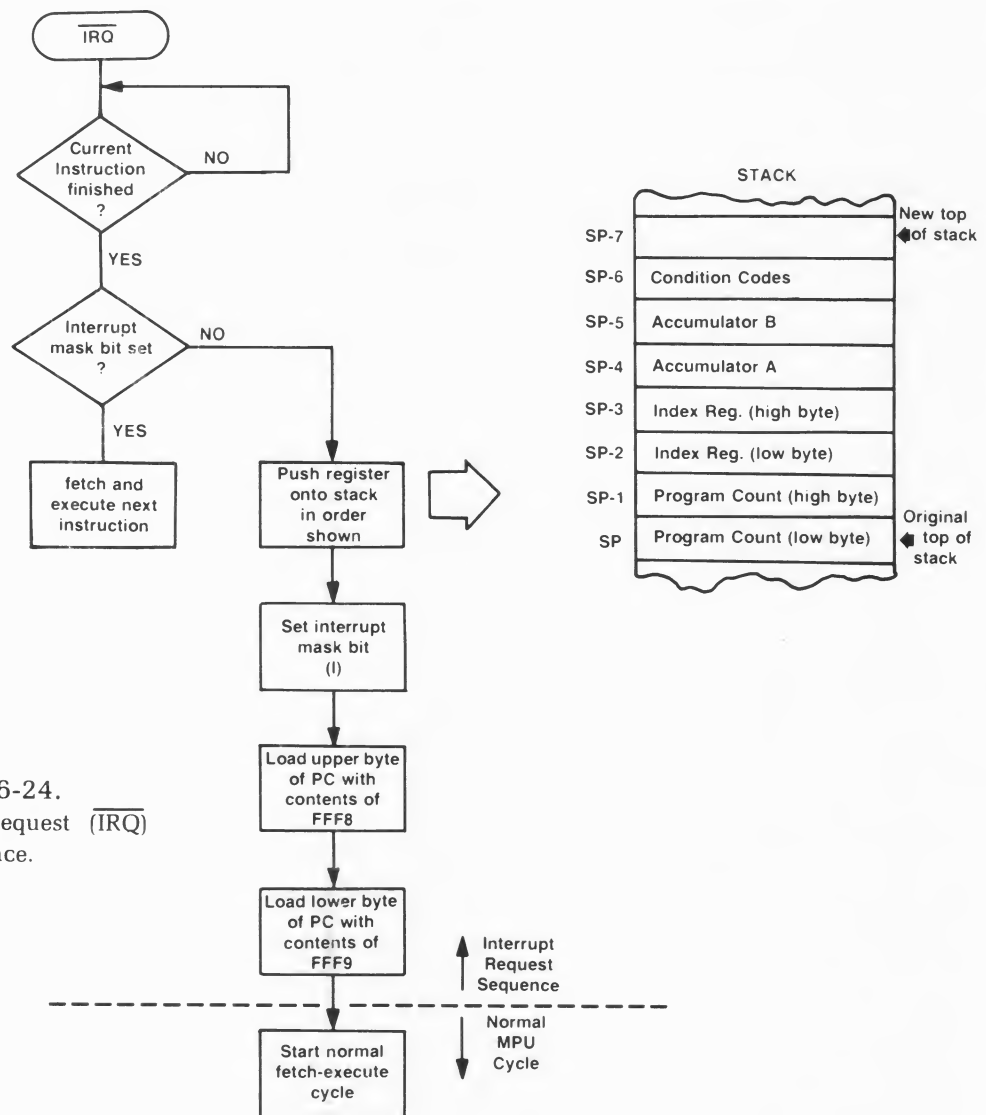


Figure 6-24.
The interrupt request ($\overline{\text{IRQ}}$)
sequence.

When the $\overline{\text{IRQ}}$ line is low, the MPU finishes the current instruction. It then checks the interrupt mask bit. If I is set to 1, the MPU ignores the interrupt request and executes the next instruction in sequence. However, if I=0, the MPU pushes the contents of the various registers onto the stack in the order shown.

Next, the interrupt mask bit is set to 1. This prevents the MPU from honoring other interrupt requests until the present interrupt has been serviced.

The address of the $\overline{\text{IRQ}}$ service routine is at addresses FFF8_{16} and FFF9_{16} . The program counter is loaded from these addresses. Thus, the next instruction to be executed will be the first instruction in the interrupt request service routine.

Once in the service routine, the MPU goes into its normal fetch-execute cycle. When the interrupt has been serviced, control can be returned to the main program by an RTI instruction.

Interrupt Mask Instructions

The 6808 MPU has two instructions that allow software control of the interrupt mask bit. You have seen that the I bit in the condition code register is set any time an interrupt sequence is initiated. This prevents an $\overline{\text{IRQ}}$ from being honored while a previous $\overline{\text{IRQ}}$ or $\overline{\text{NMI}}$ is being serviced. This is an example of setting the interrupt flag with hardware.

In many cases, it is necessary to set the interrupt flag with software. Therefore, the 6808 MPU has an instruction that can do this. It is called the "Set-Interrupt-Mask" (SEI) instruction. If you refer to your Instruction Set Summary card, you will see that this is a one-byte instruction whose opcode is 0F_{16} . The flag may be set to prevent an interruption on a part of the program that we do not wish to be interrupted. It has the effect of disabling interrupt requests.

Of course, we do not wish to permanently disable the interrupt capability. Therefore, some means must be provided for enabling the interrupt request capability. An instruction called "Clear-Interrupt-Mask" (CLI) is available for this purpose. This is a one-byte instruction whose opcode is 0E_{16} .

While we can disable or enable the interrupt request line with these instructions, they do not affect the non-maskable interrupt. As the name implies, the $\overline{\text{NMI}}$ line cannot be disabled by the I flag.

Software Interrupt (SWI) Instruction

The 6808 MPU has a software equivalent of an interrupt. It is an instruction called the "Software Interrupt" (SWI). When executed, the instruction causes the MPU to perform an interrupt sequence that is very similar to the hardware interrupt sequences already discussed. As shown on your Instruction Set Summary card, this is a one-byte instruction whose opcode is $3F_{16}$.

Figure 6-25 shows the sequence of events that occurs when this instruction is executed. First the contents of all the pertinent registers are pushed onto the stack in the order shown. Next, the interrupt mask is set so that interrupt requests cannot interfere. Finally, the software interrupt vector is obtained from addresses $FFFA_{16}$ and $FFFB_{16}$. This vector is loaded into the program counter so that the next instruction will be fetched from this address. As with the other interrupts, the MPU will return to the original program when a return-from-interrupt instruction is encountered.

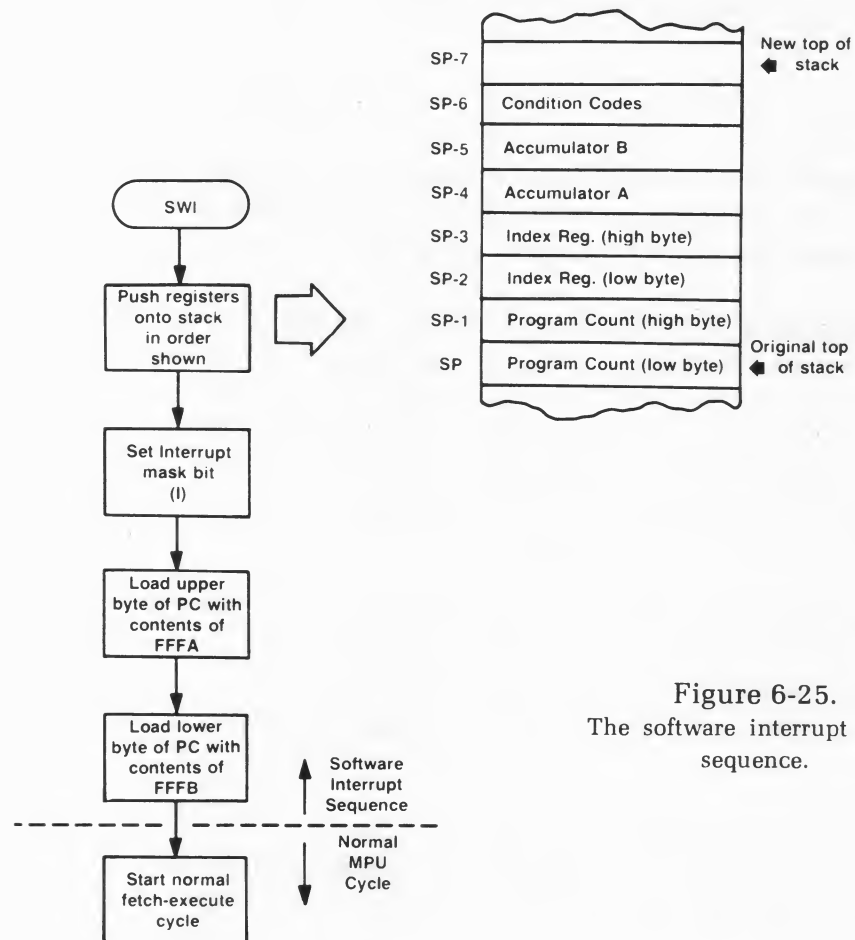


Figure 6-25.
The software interrupt (SWI) sequence.

The software interrupt instruction can be used to simulate hardware interrupts. It is also helpful for inserting pauses in a program. For example, the ET-6800 Microprocessor Trainer uses the software interrupt to perform the single-step function and to implement the breakpoint capability.

Wait for Interrupt (WAI) Instruction

One of the first instructions introduced in this course was the halt instruction (opcode $3E_{16}$). In the previous unit, you learned that this instruction is actually called a Wait-for-Interrupt (WAI). What exactly does this instruction do? It does cause the MPU to halt, but there is more to it than that.

When the WAI instruction is executed, the program counter is incremented by one. Then the contents of the program counter, index register, accumulators, and condition code register are pushed onto the stack. The order is exactly the same as if an interrupt occurs. The MPU then enters a wait state, doing nothing further until, and unless, an interrupt occurs.

The MPU can be forced back into action either by an interrupt request or by a non-maskable interrupt. The \overline{NMI} sequence is the same as that described earlier except for one important difference. Remember that the contents of the registers have already been pushed onto the stack. Thus, this part of the \overline{NMI} sequence is omitted. This allows the MPU to respond faster to the interrupt.

The \overline{IRQ} sequence is also the same as that described earlier except that the registers are not pushed onto the stack again. As always, the \overline{IRQ} signal is ignored if the interrupt mask bit is set.

Of course, the reset signal can override the wait state. Thus, there are three ways of escaping the wait state.

Self-Test Review

28. List the four types of interrupts available to the MPU.
29. Which interrupt is ignored by the MPU if the interrupt mask bit is set?
30. What is the purpose of the reset interrupt sequence?
31. From what addresses is the reset interrupt vector taken?
32. What is an interrupt vector?
33. List the sequence of events that takes place when a non-maskable interrupt occurs.

34. What is an interrupt service routine?
35. What is usually the last instruction in the interrupt service routine?
36. How does the RTI instruction affect the stack pointer?
37. Which of the interrupts does not cause data to be pushed into the stack?
38. Which instruction can be used to disable the interrupt request capability?
39. List three methods by which the MPU can be released from the wait state following the execution of a WAI instruction.
40. Under what condition will the $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ interrupts not cause data to be pushed into the stack?

Answers

- 28. Reset, non-maskable interrupt, interrupt request, and software interrupt.
- 29. Interrupt request ($\overline{\text{IRQ}}$).
- 30. To direct the MPU to the first instruction in the monitor or control program.
- 31. FFFE_{16} and FFFF_{16} .
- 32. The address of the interrupt service routine.
- 33.
 - A. The current instruction is executed.
 - B. The contents of the pertinent registers are pushed onto the stack.
 - C. The interrupt mask bit is set.
 - D. The $\overline{\text{NMI}}$ vector from addresses FFFC_{16} and FFFD_{16} is loaded into the program counter.
 - E. The instruction at the address specified by the $\overline{\text{NMI}}$ vector is fetched and executed.
- 34. A routine that takes care of the interrupt and then returns control to the main program.
- 35. The Return-From-Interrupt (RTI) instruction.
- 36. The stack pointer is incremented seven times as the previous MPU status is pulled from the stack.
- 37. Reset.
- 38. Set Interrupt Mask (SEI).
- 39.
 - A. By a reset signal.
 - B. By a non-maskable interrupt.
 - C. By an interrupt request (if $\text{I}=0$).
- 40. If the MPU is waiting for an interrupt.

EXPERIMENTS

Perform Experiments 9 and 10 in the Programming Experiment Section (Unit 7) of this course. After you finish these experiments, return to this unit and complete the unit examination.

UNIT EXAMINATION

1. If the I bit in the condition code register is set, the MPU will ignore:
 - A. The reset signal.
 - B. The non-maskable interrupt signal.
 - C. The interrupt request signal.
 - D. The software interrupt instruction.

2. Which of the following lists contains instructions that do **not** change the contents of the stack pointer?
 - A. PULA, DES, RTI, WAI.
 - B. PSHB, INS, RTS, SWI.
 - C. TXS, BSR, PULB, LDS.
 - D. PSHA, JMP, TSX, STS.

3. Which of the following program segments will successfully swap the contents of the two accumulators?

A. PSHA TAB PULA	B. PSHB TAB PULA	C. PSHA TBA PULA	D. PSHB TBA PULB
------------------------	------------------------	------------------------	------------------------

4. The stack pointer is automatically:
 - A. Decrement before data is pushed onto the stack.
 - B. Increment before data is pushed onto the stack.
 - C. Decrement after data is pushed onto the stack.
 - D. Increment after data is pushed onto the stack.

5. One difference between the JMP and JSR instruction is:
 - A. JMP can use either extended or indexed addressing.
 - B. The program count is saved when JSR is executed.
 - C. The JSR will be executed even if the interrupt mask is set.
 - D. JMP is an unconditional jump.

6. The last instruction in a subroutine is generally:
 - A. A JMP instruction.
 - B. An RTS instruction.
 - C. An RTI instruction.
 - D. A JSR instruction.

7. In the 6808 MPU, which of the following instructions could be used to transfer data from an I/O device to accumulator A?
- INPA.
 - LDAA.
 - STAA.
 - OUTA.
8. Refer to Figures 6-17 and 6-18. Which of the following program segments will read in data from the switch bank and, if the number is larger than $2A_{16}$, display it on the LED's?

A. LDAA	B. LDAA	C. LDAA	D. LDAA
80	80	80	90
00	00	00	00
CMAPA#	SUBA#	STAA	SUBA#
2A	2A	90	2A
BHI	BHI	00	BHI
01	01		01
WAI	WAI		WAI
STAA	STAA		STA
90	90		80
00	00		00

9. Which of the following types of interrupts does **not** cause data to be pushed into the stack?
- Software interrupt.
 - Non-maskable interrupt.
 - Reset interrupt.
 - Interrupt request.
10. Generally, the last instruction in an interrupt service routine will be:
- An RTI instruction.
 - An SWI instruction.
 - An RTS instruction.
 - An NMI instruction.

EXAMINATION ANSWERS

1. C — The MPU will ignore an interrupt request if the I bit of the condition code register is set.
2. D — Neither the JMP, the TSX, or the STS change the contents of the stack pointer.
3. B — B is pushed on the stack; the contents of A are transferred to B; the old contents of B are pulled into A. Thus, the contents of A and B are swapped.
4. C — The stack pointer is decremented after data is pushed into the stack.
5. B — The program count is saved when JSR is executed. It is not saved when JMP is executed.
6. B — The last instruction in a subroutine is generally an RTS instruction.

7. B — The LDAA instruction could be used to transfer data from an I/O device to accumulator A.
8. A — The data byte is loaded in accumulator A and is compared with $2A_{16}$. If it is higher than $2A_{16}$, the WAI instruction is skipped and the data is output to the LED's.
9. C — The reset interrupt does not cause data to be pushed onto the stack.
10. A — Generally, the last instruction in an interrupt sequence will be an RTI instruction.

Unit 7
PROGRAMMING EXPERIMENTS

CONTENTS

Introduction	7-3
Experiment 1. Binary/Decimal Training Program	7-4
Experiment 2. Hexadecimal/Decimal Training Program	7-11
Experiment 3. Straight Line Programs	7-19
Experiment 4. Arithmetic and Logic Instructions	7-35
Experiment 5. Program Branches	7-45
Experiment 6. Additional Instructions	7-80
Experiment 7. New Addressing Modes	7-100
Experiment 8. Arithmetic Operations	7-107
Experiment 9. Stack Operations	7-117
Experiment 10. Subroutines	7-125

INTRODUCTION

This Unit contains ten programming experiments that are to be run on your ET-6800 Microprocessor Trainer. At the end of Units 1 through 6, you will be instructed to perform one or more of these experiments.

The early programs given in this Manual are extremely simple. The later programs are more complex, but you will be able to accomplish them as you become familiar with the instruction set and programming techniques. Before you finish this course, you will be writing programs that will turn the trainer into a clock, a musical instrument, a digital voltmeter, etc.

When you complete an experiment, return to the activity guide of the unit that directed you to the experiment. This is important because you will be jumping from one point to another quite frequently.

EXPERIMENT 1

Binary/Decimal Training Program

OBJECTIVES:

To improve your ability to convert binary numbers to their decimal equivalent.

To improve your ability to convert decimal numbers to their binary equivalent.

To present the proper procedure for entering a program into the ET-6800 Microprocessor Trainer.

To demonstrate the versatility of the ET-6800 Microprocessor Trainer and microprocessors in general.

Introduction

In Unit 1, you were introduced to the binary number system. As you proceed through this course, you will find the need to convert binary numbers to decimal, and decimal numbers to binary. To improve your ability to make these conversions, you will enter a program into the ET-6800 Microprocessor Trainer to allow it to act as your instructor. In the first half of this experiment, you will use the Trainer to practice binary-to-decimal conversion.

When you use the Trainer, carefully follow all of the operating instructions. A microprocessor can only perform properly if it is programmed properly. However, you do not need programming experience at this time; just follow the instructions provided in this experiment. Do not worry about what you are entering.

The ET-6800 Trainer Manual contains a great amount of useful information in the Operation Section. You should review that section before you proceed with this experiment.

Procedure

1. Plug in the Trainer. The display should show CPU UP.
2. Push the AUTO (automatic) key. Displays H, I, N, and Z will show "prompt" characters (bottom segment of each digit illuminated), and displays V and C will show Ad. NOTE: The letters identifying each display are located near their bottom right corners.
3. Push the 0 key three times. 0's will appear in displays H, I, and N.
4. Push, but do not release the 0 key. A 0 will appear in display Z. Now release the key. The 0 will not change, but displays V and C will now show prompt characters.

NOTE: The Trainer is now ready to receive program data. If you make a data error while entering the program, do not attempt to correct the error; continue programming. Any errors will be located and corrected when you examine your program.

5. Using the Trainer keys, enter the Binary-to-Decimal training program shown in Figure 7-1. At each address specified, press the appropriate inst/data (program instruction or data) number keys (most significant number first). Displays V and C will show the inst/data word you have entered. Note that as you release the second data key, address displays H, I, N, and Z will increment (count up one), and displays V and C will show prompt characters. When you get to the end of the program, press the RESET key as indicated.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0029	08	0052	00
0001	00*	002A	08	0053	3B
0002	BD	002B	00	0054	4F
0003	FC	002C	00	0055	DB
0004	BC	002D	00	0056	BD
0005	BD	002E	80	0057	00
0006	FE	002F	BD	0058	69
0007	52	0030	FC	0059	7E
0008	5E	0031	BC	005A	00
0009	FE	0032	BD	005B	02
000A	7C	0033	FE	005C	BD
000B	00	0034	09	005D	FE
000C	01	0035	97	005E	52
000D	B6	0036	00	005F	00
000E	CO	0037	BD	0060	00
000F	03	0038	00	0061	15
0010	01	0039	69	0062	9D
0011	46	003A	5F	0063	BD
0012	25	003B	84	0064	00
0013	F6	003C	FO	0065	69
0014	CE	003D	27	0066	7E
0015	00	003E	07	0067	00
0016	00	003F	80	0068	14
0017	DF	0040	10	0069	86
0018	F2	0041	CB	006A	02
0019	BD	0042	OA	006B	CE
001A	FD	0043	4D	006C	00
001B	93	0044	26	006D	00
001C	B6	0045	F9	006E	09
001D	CO	0046	96	006F	26
001E	06	0047	00	0070	FD
001F	01	0048	84	0071	4A
0020	46	0049	0F	0072	26
0021	25	004A	1B	0073	F7
0022	F9	004B	90	0074	96
0023	BD	004C	01	0075	01
0024	FC	004D	26	0076	84
0025	BC	004E	OD	0077	3F
0026	BD	004F	BD	0078	97
0027	FE	0050	FE	0079	01
0028	52	0051	52	007A	96
				007B	00
				007C	39
					RESET

*This data may change randomly.

Figure 7-1
Binary-to-decimal training program

6. Now that you have entered the Binary-to-Decimal training program, you must examine the data for errors. Use the following sequence to examine the data and correct any errors.
 - A. Press the EXAM (examine) key. Note that the display is now asking for a 4-digit address (_ _ _ _ Ad.)
 - B. Enter the beginning address of the program (0000). As soon as the last address digit is entered, displays V and C show the contents of that memory location. NOTE: The address is a memory location in the Trainer.
 - C. Now compare the displayed address and data with the address and inst/data columns in the program.
 - D. If the displayed data is incorrect, press the CHAN (change) key. The data displays will now show prompt characters. Enter the correct data.
 - E. Press the FWD (forward) key. The address will increment and the data for that memory location will be shown. Correct the data if necessary.
 - F. Continue to step through the program with the FWD key, and correct data as necessary, until you reach the end of the program. It is not necessary to examine or modify the memory beyond address 007C since it will have no effect on the program.
7. Press the RESET key.
8. Press the DO key, then enter address 0002. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 6 through 8.
9. Press the F key. A 6-bit binary number should appear in the display. This is a random number and should change in value when you are told to "GO" next time.
10. Examine the binary number and determine its decimal value. Then press the D key. Two prompt characters should appear in the display.

11. Enter the decimal value of the binary number previously displayed (most significant digit first.) For values less than 10, enter a 0 before you enter the value. After a short period of time, the Trainer will indicate whether or not your answer is correct.
12. If your answer was correct, the Trainer will display YES. A moment later, the word GO will replace the decimal number.

If your answer was incorrect, the Trainer will display NO. The same binary number will again be displayed. Determine and enter the decimal value as described in steps 10 and 11.

13. Refer again to steps 9 through 12 and practice converting binary numbers to their decimal equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

Discussion

Now that you have used the Trainer and its microprocessor, you have accomplished three objectives. First, you are becoming proficient in binary-to-decimal conversion. Second, you have been introduced to the correct method for entering, examining, and modifying a program. Third, you have been shown how a simple set of instructions can produce a powerful training aid. However, you should remember, a microprocessor can only perform what you tell it. One incorrect instruction can produce totally unexpected results.

Now, reprogram the Trainer for decimal-to-binary instruction. Since you will be using the same memory locations used in the first half of this experiment, the Binary-To-Decimal program will disappear.

Procedure (continued)

14. Press the RESET key.
15. Press the AUTO key, and enter address 0000.
16. Using the Trainer keys, enter the Decimal-to-Binary training program shown in Figure 7-2.
17. Now that you have entered the Decimal-to-Binary program, press the EXAM key and enter address 0000.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0033	4F	0066	00
0001	CE	0034	E6	0067	00
0002	C1	0035	00	0068	80
0003	6F	0036	C5	0069	7E
0004	BD	0037	10	006A	00
0005	FE	0038	27	006B	01
0006	50	0039	03	006C	BD
0007	5E	003A	AB	006D	FE
0008	FE	003B	03	006E	52
0009	96	003C	19	006F	15
000A	00	003D	56	0070	1D
000B	8B	003E	24	0071	00
000C	01	003F	03	0072	00
000D	19	0040	AB	0073	00
000E	81	0041	06	0074	80
000F	63	0042	19	0075	BD
0010	23	0043	08	0076	00
0011	01	0044	8C	0077	7E
0012	4F	0045	00	0078	BD
0013	97	0046	88	0079	FC
0014	00	0047	26	007A	BC
0015	B6	0048	EB	007B	7E
0016	CO	0049	BD	007C	00
0017	03	004A	00	007D	1C
0018	01	004B	7E	007E	36
0019	46	004C	BD	007F	BD
001A	25	004D	FC	0080	00
001B	ED	004E	BC	0081	8E
001C	96	004F	D6	0082	32
001D	00	0050	00	0083	01
001E	BD	0051	11	0084	39
001F	FE	0052	26	0085	00
0020	20	0053	18	0086	00
0021	B6	0054	BD	0087	00
0022	CO	0055	FE	0088	32
0023	06	0056	52	0089	08
0024	01	0057	00	008A	02
0025	46	0058	00	008B	16
0026	25	0059	00	008C	04
0027	F9	005A	3B	008D	01
0028	BD	005B	4F	008E	86
0029	FC	005C	DB	008F	02
002A	BC	005D	BD	0090	CE
002B	C6	005E	00	0091	00
002C	03	005F	7E	0092	00
002D	CE	0060	CE	0093	09
002E	00	0061	C1	0094	26
002F	85	0062	3F	0095	FD
0030	BD	0063	BD	0096	49
0031	FD	0064	FE	0097	26
0032	25	0065	50	0098	F7
				0099	39
					RESET

*This data may change randomly

Figure 7-2
Decimal-to-binary training program.

18. Using the FWD key, compare the Trainer memory contents with the program address and inst/data listing. If you must correct any data, press the CHAN key and enter the proper data.
19. After you have checked the program, press the RESET key.
20. Press the DO key, then enter address 0001. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 17 through 20.
21. Press the F key. A 2-digit decimal number should appear in the display, next to the word GO. This is a random number and should change in value when you are told to "GO" next time.
22. Examine the decimal number and determine its binary value. Then press the D key. Six prompt characters should appear in the display.
23. Enter the binary value of the decimal number previously displayed, beginning with the most significant bit (MSB). If the decimal value is less than 32, be sure to enter any leading zeros. NOTE: Although the program will accept any number combination, you should use only 1's and 0's.
24. If your answer was correct, the Trainer will display YES a short time after you enter the last binary bit. A moment later, the Trainer will display GO.

If your answer was incorrect, the Trainer will display NO a short time after you enter the last binary bit. A moment later, the same decimal number will be displayed again. Determine and enter the binary value as described in steps 22 and 23.
25. Refer again to steps 21 through 24 and practice converting decimal numbers to their binary equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

Discussion

In this half of the experiment, you were given further experience in programming with the ET-6800 Microprocessor Trainer. You also improved your ability to readily translate decimal numbers into binary. This ability will become very useful as you progress through the Microprocessor Course.

EXPERIMENT 2

Hexadecimal/Decimal Training Program

OBJECTIVES: *To practice the conversion of decimal numbers to their hexadecimal equivalent.*

To practice the conversion of hexadecimal numbers to their decimal equivalent.

Introduction

Binary numbers are used in all microprocessors to represent data and instructions. But binary numbers are difficult to work with . . . especially when the number contains 8_{10} bits or more. To simplify programming, microprocessor designers usually use other number systems, like octal or hexadecimal, to represent binary data. Both octal and hexadecimal are just shorthand notations of binary numbers. Although the numbers are entered in hexadecimal or octal, the microprocessor “sees” them as binary. This simplifies programming.

For example, the binary number 10011111_2 requires eight key closures for entry. Fortunately, this same number can be represented in hexadecimal as $9F_{16}$ and requires only two key closures for entry. Fewer key closures means less programming errors and more efficient programming.

Your Microprocessor Trainer is based on the hexadecimal number system. You probably noticed this when you loaded the programs in the previous experiment; all instructions were coded in hexadecimal. The Microprocessor Trainer normally displays data in hexadecimal form. Of course, special programs allow the Trainer to accept binary or decimal numbers, as you saw in the first experiment. However, these special programs waste a portion of the microprocessors potential power and aren't necessary because you can make the conversion from decimal to hexadecimal with a little practice. That's the purpose of this experiment . . . to sharpen your conversion skills.

Again, you will use the Microprocessor Trainer for this purpose. First, you'll enter a program that allows you to practice conversion from decimal to hexadecimal. Then you'll load the second program that reverses the process. You'll find that it's not as difficult as it might appear.

18

12 8 4 2 1
1 0 0 1 0

Now briefly review decimal-to-hexadecimal conversion. Initially, it's helpful to make up a chart of decimal numbers and their hexadecimal equivalents, as shown here.

DECIMAL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEXADECIMAL	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Recall that hexadecimal is a base 16_{10} number system. Both systems use identical numbers from 0 through 9. However, at decimal number 10, the hexadecimal system shifts to characters of the alphabet, as shown by the letters A through F. Conversion of a decimal number to its hexadecimal equivalent is a simple process where the decimal number is repeatedly divided by 16_{10} , with the remainder producing the equivalent hexadecimal number. This example will use only 2-digit numbers, since that's what you'll be converting in this experiment.

Suppose you want to convert 92_{10} to hexadecimal. The first step is to divide 92_{10} by 16_{10} as shown below.

$$\begin{array}{r}
 5 \\
 16 \overline{) 92} \\
 \underline{- 80} \\
 12
 \end{array}$$

Remainder $12_{10} = C_{16} \leftarrow \text{LSD}$

The quotient is 5, but remember, we aren't concerned with this at the moment. We're interested in the remainder, in this case 12_{10} , because it forms the LSD of the equivalent hexadecimal number. Now, refer to the chart and find that $12_{10} = C_{16}$ and write this down as the LSD of the hex equivalent. The next step is to take the quotient of the previous division, in this case 5, and divide it by 16_{10} , as shown below.

$$\begin{array}{r}
 0 \\
 16 \overline{) 5} \\
 \underline{- 0} \\
 5
 \end{array}$$

Remainder $\rightarrow 5_{10} = 5_{16} \leftarrow \text{MSD}$

Of course, the quotient of this division is 0, signifying that the remainder, 5_{10} , is the MSD of the hexadecimal number. Checking the chart, you find that $5_{10} = 5_{16}$. Combining the MSD (5_{16}) and LSD (C_{16}), you find that the hex equivalent of 92_{10} is $5C_{16}$. You'll find that, after you've made the conversion a few times, you'll be able to do them in your head. You'll get that practice in this experiment.

Procedure

1. Plug in the Trainer and press the RESET key.
2. Press AUTO and then enter address 0000.
3. Now enter the Decimal-to-Hexadecimal training program, shown in Figure 7-3, into the Trainer. When you've entered the last program instruction press the RESET key as shown at the end of the program.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0024	BD	0048	52	006C	00
0001	CE	0025	FE	0049	00	006D	00
0002	C1	0026	52	004A	3B	006E	00
0003	6F	0027	08	004B	4F	006F	00
0004	BD	0028	08	004C	DB	0070	00
0005	FE	0029	00	004D	BD	0071	80
0006	50	002A	00	004E	00	0072	39
0007	5E	002B	00	004F	63	0073	86
0008	FE	002C	80	0050	7E	0074	02
0009	96	002D	BD	0051	00	0075	CE
000A	00	002E	FC	0052	01	0076	00
000B	8B	002F	BC	0053	BD	0077	00
000C	01	0030	BD	0054	FE	0078	09
000D	19	0031	FE	0055	52	0079	26
000E	97	0032	09	0056	00	007A	FD
000F	00	0033	36	0057	00	007B	4A
0010	B6	0034	4F	0058	15	007C	26
0011	CO	0035	D6	0059	9D	007D	F7
0012	03	0036	00	005A	BD	007E	39
0013	46	0037	CO	005B	00		RESET
0014	25	0038	10	005C	63		
0015	F3	0039	25	005D	BD		
0016	96	003A	04	005E	FC		
0017	00	003B	8B	005F	BC		
0018	BD	003C	0A	0060	7E		
0019	FE	003D	20	0061	00		
001A	20	003E	F8	0062	16		
001B	B6	003F	CB	0063	36		
001C	CO	0040	10	0064	BD		
001D	06	0041	1B	0065	00		
001E	46	0042	33	0066	73		
001F	25	0043	11	0067	32		
0020	FA	0044	26	0068	01		
0021	BD	0045	OD	0069	BD		
0022	FC	0046	BD	006A	FD		
0023	BC	0047	FE	006B	8D		

* This data may change randomly

Figure 7-3

Decimal-to-Hexadecimal training program

4. Check the stored program by first pressing the EXAM key, and then entering address 0000. Now use the FWD key to step through the program, comparing the contents of memory with the program in Figure 7-3. Remember, the four left-most digits of the display represent the memory address and the two digits at the right are the contents of memory that should correspond with the INST/DATA listing of the program. If you find a mistake, correct it by first pressing the CHAN key and then entering the proper data.
5. When you're satisfied that the program is correct, press the RESET key.
6. Now it's time to execute the program. Do this by pressing the DO key and then entering address 0001. The word "GO" should now appear in the two left-most digits. If the display is blank, or if other numbers or letters appear, there is an error in the program and steps 4 and 5 should be repeated.
7. Now press the F key. A 2-digit "decimal" number will appear on the display. The Trainer is asking you to convert this decimal number to its hexadecimal equivalent. Therefore, examine the decimal number and then convert it to hexadecimal.
8. Enter your answer by first pressing key D. Two prompt characters will appear in the left-most digits. Now enter your hexadecimal number.

If you respond correctly, the Trainer will display "YES" for a short period and then give you another "GO." Pressing the F key will cause another random number to be displayed.

An incorrect response will result in the word "NO" on the display. After a short delay, the original decimal number will reappear and you should try the conversion process again. This cycle continues until you arrive at the correct answer.

9. Repeat steps 7 and 8, practicing conversion until you're confident of your ability. A good guideline to follow is . . . when you answer 10 consecutive queries correctly, you're probably proficient.

Discussion

As you worked through the exercises in this experiment, you probably developed your own shorthand method of conversion. After a few queries, you probably found that you didn't need the decimal-to-hexadecimal conversion chart any longer . . . you had the chart committed to memory. Perhaps you noticed that when 16_{10} is divided into the 2-digit decimal numbers used in this experiment, the resulting quotient always equals the MSD of the hexadecimal equivalent. Naturally, the remainder is the LSD. However, this only works for decimal numbers less than 159_{10} . For larger numbers, the procedure studied earlier must be used.

Since the Microprocessor Trainer displays data in hexadecimal and we naturally think in decimal, the conversion process must be reversed to interpret output data from the Trainer. For example, if the Trainer is programmed to add the numbers $1A_{16}$ and $9B_{16}$, the result $B5_{16}$ will be displayed. This hexadecimal number means very little. To understand the result, you must convert the sum ($B5_{16}$) to its decimal equivalent (181_{10}). Now the answer is clear.

Several methods can be used to change hexadecimal numbers to decimal. One process uses double conversion; first, the hexadecimal number is reduced to its binary equivalent; next, the resulting binary number is transformed into the resulting decimal equivalent.

Another, more commonly used method, is to use positional notation, inherent in any number system, and multiply each digit by its weighted value and then add the products. For example, the decimal equivalent of the hex number 11_{16} is derived as shown below:

Assign Weights:	16^1	16^0	Positional Weights
	1	1	
Weight \times Digit:	$1 \times 16^1 = 16$	$1 \times 16^0 = 1$	
Add Products:	$16 + 1 = 17$		
Final Result:	$11_{16} = 17_{10}$		

The first step is to assign positioned weights to each digit. Since the number is hexadecimal, each position represents a power of 16_{10} . Next, multiply each digit by its positional weight. Finally, add the products. The resulting sum is the decimal equivalent. Therefore, as shown in the example, 11_{16} is equal to 17_{10} .

Now try a problem that's a bit more difficult . . . converting $6B_{16}$ to decimal. To begin with, this expression hardly looks like a number. Instead, it's a combination of a number and a letter. However, the notation at the bottom of the expression denotes a base 16 number so we know it's hexadecimal. The translation process is almost identical to the previous example. The only difference being that the hexadecimal "letter" must be changed to decimal before it can be multiplied by the positional weight. The conversion process is shown below.

Assign Weights:

	16^1	16^0	
	6	B	
Convert to Decimal:	$6_{16} = 6_{10}$	$B_{16} = 11_{10}$	
Weight \times Digit:	$6 \times 16^1 = 96$	$11 \times 16^0 = 11$	
Add Products:	$96 + 11 = 107$		
Final Result:	$6B_{16} = 107_{10}$		

Again, we begin by assigning positional weights to each digit. However, now the second step is to convert the hexadecimal characters to decimal numbers. Recall that 6_{16} is equal to 6_{10} and that B_{16} equals 11_{10} . Now multiply the weight by the decimal numbers, add the products and obtain the final result. As shown, the decimal equivalent of $6B_{16}$ is 107_{10} .

In the next section of this experiment, you will load a hexadecimal-to-decimal training program in the Trainer and then practice hexadecimal-to-decimal conversion.

Procedure (Continued)

10. Prepare to enter the new program by pressing the RESET key. Next press the AUTO key and then enter address 0000.
11. Refer to Figure 7-4 and enter the Hexadecimal-to-Decimal training program listed there. When you've entered all of the instructions, press the RESET key as indicated at the end of the program.
12. Check the program that you've loaded by pressing the EXAM key and then entering address 0000. Use the FWD key to step through the program, comparing the stored program with the program listing in Figure 7-4. Use the CHAN key to correct any errors that you find.

When you are satisfied that the program is correct, press the RESET key.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00	0024	BD	0048	FE	006C	8D
0001	CE	0025	FC	0049	52	006D	00
0002	C1	0026	BC	004A	00	006E	00
0003	6F	0027	BD	004B	3B	006F	00
0004	BD	0028	FE	004C	4F	0070	00
0005	FE	0029	52	004D	DB	0071	00
0006	50	002A	08	004E	BD	0072	80
0007	5E	002B	08	004F	00	0073	39
0008	FE	002C	00	0050	64	0074	86
0009	96	002D	00	0051	7E	0075	02
000A	00	002E	00	0052	00	0076	CE
000B	4C	002F	80	0053	01	0077	00
000C	81	0030	BD	0054	BD	0078	00
000D	63	0031	FC	0055	FE	0079	09
000E	23	0032	BC	0056	52	007A	26
000F	01	0033	BD	0057	00	007B	FD
0010	4F	0034	FE	0058	00	007C	4A
0011	97	0035	09	0059	15	007D	26
0012	00	0036	5F	005A	9D	007E	F7
0013	B6	0037	80	005B	BD	007F	39
0014	CO	0038	10	005C	00		RESET
0015	03	0039	25	005D	64		
0016	46	003A	04	005E	BD		
0017	25	003B	CB	005F	FC		
0018	FO	003C	OA	0060	BC		
0019	96	003D	20	0061	7E		
001A	00	003E	F8	0062	00		
001B	BD	003F	8B	0063	19		
001C	FE	0040	10	0064	36		
001D	20	0041	1B	0065	BD		
001E	B6	0042	D6	0066	00		
001F	CO	0043	00	0067	74		
0020	06	0044	11	0068	32		
0021	46	0045	26	0069	01		
0022	25	0046	OD	006A	BD		
0023	FA	0047	BD	006B	FD		

Figure 7-4
Hexadecimal-to-decimal training program

13. Now execute the program by first pressing the DO key and then entering address 0001. The word "GO" should appear on the display. The absence of this word indicates a programming error and you should go back and recheck the program as outlined in step 12.
14. Now press the F key. A 2-digit "hexadecimal" number will appear. The Trainer is asking for the decimal equivalent of this number. Convert the hexadecimal number into its decimal equivalent. Then enter your answer by pressing the D key. Two prompt characters will appear. Now enter your answer.

If your response is correct, the Trainer will display "YES." You can then continue these conversion exercises by again pressing the F key.

However, if your answer is incorrect, the Trainer will display "NO." After a short delay, the original hexadecimal number will reappear, and you can try again.

15. Continue the conversion training program until you are confident of your ability to change hexadecimal numbers to decimal numbers. The standard of ten correct conversions in a row is a good guideline.

Discussion

The translation of hexadecimal numbers into decimal equivalent numbers is an important part of your training.

You will find this skill is extremely handy when you begin to write programs later in this course. Now you should be able to convert between hexadecimal and decimal numbers with ease. Perhaps you even developed your own shorthand methods for these translations. If so, use them. However, a word of caution . . . be sure they work for all numbers. As mentioned previously, some techniques work with small numbers, but not with large numbers.

EXPERIMENT 3

Straight Line Programs

OBJECTIVES:

To demonstrate the instructions presented in Unit 2 with simple programs.

To present three new instructions and use them in simple programs.

To demonstrate some programming pitfalls.

To demonstrate the difference between RAM and ROM.

Introduction

Unit 2 introduced you to the basic microprocessor and its internal structure. You also learned six basic microprocessor instructions that are represented by 8-bit binary numbers called "op codes." Op codes allow you to use the microprocessor for data manipulation. Figure 7-5 lists the six instructions and their op codes. It also lists three new instructions that you will use in this experiment. These new instructions use the inherent addressing mode described in Unit 2.

This is the first experiment to introduce microprocessor instructions that you can identify. There are a number of Trainer keyboard commands that you must learn in order to examine and use the microprocessor instructions. The Trainer commands that you should know for this experiment are:

DO — Execute the program, beginning at the address specified after this key is pressed.

EXAM (examine) — Display the address and memory contents at the address specified after this key is pressed. Memory contents can be changed by pressing the **CHAN** key and entering new data.

FWD (forward) — Advance to the next memory location and display the contents.

CHAN (change) — Open the memory location being examined so that new data can be entered.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator (Immediate)	LDA	1000 0110 ₂ or 86 ₁₆	Load the contents of the next memory location into the accumulator.
Add (Immediate)	ADD	1000 1011 ₂ or 8B ₁₆	Add the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Load Accumulator (Direct)	LDA	1001 0110 ₂ or 96 ₁₆	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add (Direct)	ADD	1001 1011 ₂ or 9B ₁₆	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator (Direct)	STA	1001 0111 ₂ or 97 ₁₆	Store the contents of the accumulator in the memory location whose address is given by the next byte.
Halt (Inherent)	HLT	0011 1110 ₂ or 3E ₁₆	Stop all operations.
Clear Accumulator (Inherent)	CLRA	0100 1111 ₂ or 4F ₁₆	Reset all bits in the accumulator to 0.
Increment Accumulator (Inherent)	INCA	0100 1100 ₂ or 4C ₁₆	Add 1 to the contents of the accumulator.
Decrement Accumulator (Inherent)	DECA	0100 1010 ₂ or 4A ₁₆	Subtract 1 from the contents of the accumulator.

Figure 7-5
Instructions used in Experiment 3.

BACK — Go back to the previous memory location and display the contents.

AUTO (automatic) — Open the memory location specified, after this key is pressed, so that data can be entered. After data has been entered, automatically advance to the next memory location and wait for data.

SS (single step) — Go to the address specified by the program counter and execute the instruction at that address. Wait at the next instruction.

ACCA (accumulator) — Display the contents of the accumulator when this key is pressed. Accumulator contents can be changed by pressing the CHAN key and entering new data.

PC (program counter) — Display the contents of the program counter. This points to the next location in memory that the microprocessor will “fetch” from. Program counter contents can be changed by pressing the CHAN key and entering the new address.

RESET — Clear any Trainer keyboard commands and display “CPU UP.” Memory contents and microprocessor contents are not disturbed.

You have access to all of these keyboard commands after the RESET key is pressed.

In this experiment, you will load some simple straight-line programs into the Trainer and examine how the microprocessor executes them. In its normal mode of operation, the microprocessor executes programs much too fast for a person to follow. It can execute hundreds of thousands of instructions each second. To allow us to witness the operation of the MPU, this high speed operation must be slowed down. The Microprocessor Trainer has a mode of operation that allows us to control the execution of single instructions. In this single-step mode, we can look at the contents of the accumulator, the program counter, and various memory locations, after each instruction is executed. In this way, we can follow exactly how the computer performs each step of the program. For this reason, you will use the single-step mode for most of the programs in this experiment.

Procedure

1. Switch your Trainer on, and press the RESET key.
2. Your first program will use the immediate addressing mode to add two numbers. Press AUTO and enter starting address 0000. Then load the hex contents of the program listed in Figure 7-6.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	21	33 ₁₀	Operand 1.
0002	8B	ADD	Add to accumulator immediately with
0003	17	23 ₁₀	Operand 2.
0004	3E	HLT	Stop.

Figure 7-6

Addition of two numbers through the immediate addressing mode.

3. Press the RESET key, then examine your program to make sure it was properly entered. **Always** examine your program after it is entered.
4. Press the ACCA key and record the value 1 3. This is a random number since no data has been loaded.
5. Press the PC key, then change the contents of the program counter to 0000 (the starting address of your program).
6. Press the SS key. This lets the Trainer execute the first instruction. The display should show 00028b. 0002 represents the address of the next instruction; 8b is the next instruction.
7. Press the ACCA key and record the value 2 1. The first program instruction was LDA, and the next byte contained the data (operand) to be loaded, which is 21₁₆. This should be the value you recorded in this step.
8. Press the PC key and record the value 000 2. This value points to the next memory location, which should be 0002.

You may have noted that the address 0002 and instruction 8b were displayed when you first pressed the SS key. This would seem to indicate that 8b was already fetched and the program counter should point to address 0003. However, the control program allows the Trainer to “look” at the next instruction.

9. Press the SS key and record the value _ _ _ _ _ . The second instruction has been executed and the display should show the next instruction and its address.
10. Press the ACCA key and record the value _ _ . The second operand has been added to the first operand and the sum is stored in the accumulator.
11. Press the SS key. Note that the display does not change. This is because the next instruction was a halt instruction ($3E_{16}$). The Trainer is preprogrammed to stop at a halt instruction. It also loses control of the single-step function when the halt instruction is implemented.
12. Enter the program (HEX contents) listed in Figure 7-7. Then examine the program to make sure it is properly entered.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	07	07_{16}	operand 1 which is stored at this address.
0002	9B	ADD	Add to accumulator direct with operand 2
0003	08	08_{16}	which is stored at this address.
0004	97	STA	Store the sum
0005	09	09_{16}	at this address.
0006	3E	HLT	Stop.
0007	20	32_{10}	Operand 1.
0008	17	23_{10}	Operand 2.
0009	00	00	Reserved for sum.

Figure 7-7

Addition of two numbers through the direct addressing mode.

13. Press the ACCA key and record the value 21 . This is the value obtained in the previous program, a value you entered prior to this program, or a random value produced when you plugged in the Trainer.

14. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step.

Step 1 display 0002 9b

ACCA 2 0

Step 2 display 0004 97

ACCA 3 7

Step 3 display 0006 3E

ACCA 5 7

15. Examine address 0009. Its value is . This value should be identical to the value now stored in the ACCA.
16. Now compare your recorded data with the program in Figure 7-7. This will give you a general picture of how the microprocessor uses various instructions and data to perform a desired function.
17. Change the data in the ACCA and at address 0009 to FF, then execute the program with the DO key. This is done by depressing the DO key and then entering the address of the first instruction (0000). This allows the MPU to execute the program at its normal speed. After the program runs, you must press RESET to return control to the keyboard.
18. The data in the ACCA is and the data in address 0009 is . These should be the same and equal to the sum of the two operands.
19. The program counter contains the address . This should be the address of the next memory location after the HLT instruction.

20. Now write a program of your own. Using the **direct** addressing mode, write a program that will multiply 4 times 4, by adding 4 to itself in three consecutive steps. The final answer should be held in the accumulator. After you write your program, enter it into the Trainer and execute it. Keep trying until it produces a final result of 10_{16} (which is 16_{10}) in the accumulator.

One solution to the problem is shown in Figure 7-8. Yours should be similar, although not necessarily identical.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/DECIMAL CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	09	09_{16}	operand 1 which is stored at this address.
0002	9B	ADD	Add to accumulator direct with
0003	09	09_{16}	operand 1 which is stored at this address.
0004	9B	ADD	Add to accumulator direct with
0005	09	09_{16}	operand 1 which is stored at this address.
0006	9B	ADD	Add to accumulator direct with
0007	09	09_{16}	operand 1 which is stored at this address.
0008	3E	HLT	Stop.
0009	04	04_{10}	Operand 1.

Figure 7-8

Multiplication of a number by another through multiple addition in the direct addressing mode.

21. Load the program shown in Figure 7-8 into the Trainer. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _ . ACCA _ _ .

Step 2 display _ _ _ _ _ . ACCA _ _ .

Step 3 display _ _ _ _ _ . ACCA _ _ .

Step 4 display _ _ _ _ _ . ACCA _ _ .

22. According to the microprocessor, the product of 4_{16} times 4_{16} is $-_{16}$.
23. Now that you are becoming acquainted with the instructions described in Unit 2, examine the three instructions introduced in this Experiment. Enter the program listed in Figure 7-9.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear accumulator.
0001	97	STA	Store the contents
0002	0A	$0A_{16}$	at this address.
0003	4C	INCA	Increment accumulator.
0004	97	STA	Store the contents
0005	0B	$0B_{16}$	at this address.
0006	4A	DECA	Decrement accumulator.
0007	97	STA	Store the contents
0008	0C	$0C_{16}$	at this address.
0009	3E	HLT	Stop.
000A	FF	FF_{16}	Reserved for data.
000B	FF	FF_{16}	Reserved for data.
000C	FF	FF_{16}	Reserved for data.

Figure 7-9

Implementation of the Clear, Increment, and Decrement instructions.

24. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display	000197.	ACCA 00.
Step 2 display	---34C.	ACCA 00.
Step 3 display	---401.	ACCA 01.
Step 4 display	---64A.	ACCA 01.
Step 5 display	---791.	ACCA 00.
Step 6 display	---93E.	ACCA --.

25. Compare your accumulated data with the program in Figure 7-9. Note that when op codes $4F_{16}$, $4C_{16}$, and $4A_{16}$ are executed, the single-step display advances only one address location. This is because of their inherent addressing mode; immediate and direct addressing modes require two locations in memory.
26. In the Unit Examination, you were shown a program to swap the contents of two memory locations. Now examine the process using the Trainer. Enter the program listed in Figure 7-10.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with operand 1
0001	10	10_{16}	stored at this address.
0002	97	STA	Store operand 1
0003	12	12_{16}	at this address.
0004	96	LDA	Load accumulator direct with operand 2
0005	11	11_{16}	stored at this address.
0006	97	STA	Store operand 2
0007	10	10_{16}	at this address.
0008	96	LDA	Load accumulator direct with operand 1
0009	12	12_{16}	stored at this address.
000A	97	STA	Store operand 1
000B	11	11_{16}	at this address.
000C	4F	CLRA	Clear the accumulator.
000D	97	STA	Store the contents
000E	12	12_{16}	at this address.
000F	3E	HLT	Stop.
0010	AA	170_{10}	Operand 1.
0011	BB	187_{10}	Operand 2.
0012	00	00	Temporary storage.

Figure 7-10
Data transfer between two addresses.

27. Set the program counter to starting address 0000 and single-step through the program. Record the specified information after each step.

Step 1 display	-- 297.	ACCA	A A .
Step 2 display	-- 496.	ACCA	✓ .
Step 3 display	-- 697.	ACCA	b b .
Step 4 display	-- 896.	ACCA	✓ .
Step 5 display	-- A97.	ACCA	A A .
Step 6 display	-- C4F.	ACCA	✓ .
Step 7 display	-- 297.	ACCA	0 0 .
Step 8 display	-- E3E.	ACCA	0 0 .

28. Examine address:

0010 ~~2~~ ~~b~~.

0011 ~~A~~ ~~A~~.

0012 ~~0~~ ~~0~~.

29. Compare your accumulated data with the program in Figure 7-10.

30. Now you will examine some common programming pitfalls. Without modifying the previous program, except as directed in Figure 7-11, enter the program listed in Figure 7-11.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with operand 1.
0001	4F	79 ₁₀	
0002	97	STA	Store operand 1 at this address.
0003	05	05 ₁₆	
0004	4A	DECA	Decrement accumulator Stop.
0005	3E	HLT	

Figure 7-11

Storing data at an address in the program.

31. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _.	ACCA _ _.
Step 2 display _ _ _ _ _.	ACCA _ _.
Step 3 display _ _ _ _ _.	ACCA _ _.
Step 4 display _ _ _ _ _.	ACCA _ _.
Step 5 display _ _ _ _ _.	ACCA _ _.
Step 6 display _ _ _ _ _.	ACCA _ _.
Step 7 display _ _ _ _ _.	ACCA _ _.
Step 8 display _ _ _ _ _.	ACCA _ _.
Step 9 display _ _ _ _ _.	ACCA _ _.

32. Compare your accumulated data with the program in Figure 7-11. Note that the data in the accumulator (operand 1) has been stored at address 0005. This removed the HLT instruction and allowed the microprocessor to continue executing any valid instructions in memory. In this case, the remaining unaltered instructions from the previous program are used. When you write a program, **make sure** you do not store data at an address that contains a needed instruction or data.
33. Using the data you accumulated in step 31 of this experiment, plus the programs listed in Figures 7-10 and 7-11, determine the contents of address:

0010 _ _.

0011 _ _.

0012 _ _.

34. Now examine the Trainer contents at address:

0010 _ _.

0011 _ _.

0012 _ _.

Your estimated data from step 33, and the actual contents should be identical. If they are not, re-examine your calculations and the contents of each memory location from 0000 to 0012. You might have inadvertently modified the contents of an address in the previous steps.

35. Without modifying the previous program, except as directed in Figure 7-12, enter the program listed in Figure 7-12.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	40	64 ₁₀	operand 1.
0002	8B	ADD	Add to accumulator immediately with
0003	0A	10 ₁₀	operand 2.
0004	97	STA	Store the sum
0005	07	07 ₁₆	at this address.
0006	4F	CLRA	Clear accumulator.
0007	00	00	Reserved for data.

Figure 7-12

Addition of two numbers with immediate addressing.

36. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display -----.	ACCA ---.
Step 2 display -----.	ACCA ---.
Step 3 display -----.	ACCA ---.
Step 4 display -----.	ACCA ---.
Step 5 display -----.	ACCA ---.
Step 6 display -----.	ACCA ---.
Step 7 display -----.	ACCA ---.
Step 8 display -----.	ACCA ---.
Step 9 display -----.	ACCA ---.

37. Compare your accumulated data with the program in Figure 7-12. Note that the Trainer executed the instructions beyond address 0007. This occurred because there was no halt instruction in the program. Always end your program with a halt instruction. If you don't, the microprocessor will try to execute all of the information contained in memory, thinking it is part of the program. In the process, the program you entered may get modified.
38. This final programming pitfall illustrates a problem almost everybody experiences. Enter the program listed in Figure 7-13.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	07	07 ₁₆	operand 1 stored at this address.
0002	8B	ADD	Add to accumulator direct with
0003	07	07 ₁₆	operand 1 stored at this address.
0004	8B	ADD	Add to accumulator direct with
0005	07	07 ₁₆	operand 1 stored at this address.
0006	3E	HLT	Stop.
0007	05	05 ₁₀	Operand 1.

Figure 7-13

Multiplication of two numbers using successive addition in the direct addressing mode.

39. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display _ _ _ _ _ . ACCA _ _ .

Step 2 display _ _ _ _ _ . ACCA _ _ .

Step 3 display _ _ _ _ _ . ACCA _ _ .

40. Compare your accumulated data with the program in Figure 7-13. The program should have added 05 three times (5×3) for the answer 0F. The Trainer indicates the answer is 13. This discrepancy occurred because the program contains the wrong **addressing mode op code** for the ADD function. It should be 9B rather than 8B. Return to Figure 7-13 and change the two ADD op codes to 9B so the program will be correct.
41. In Unit 2, you were shown that RAM (random access memory) was a read/write type memory, while ROM (read only memory) is a preprogrammed memory that can only be read and not written into. To examine these memory types, enter FF at address 0000 through 000F.
42. Examine the following memory locations and write down the contents next to each address. Use the first data column for each address. You will use the second column later.

ADDRESS	DATA	DATA	ADDRESS	DATA	DATA
0000	--	--	FD00	--	--
0001	--	--	FD01	--	--
0002	--	--	FD02	--	--
0003	--	--	FD03	--	--
0004	--	--	FD04	--	--
0005	--	--	FD05	--	--
0006	--	--	FD06	--	--
0007	--	--	FD07	--	--
0008	--	--	FD08	--	--
0009	--	--	FD09	--	--
000A	--	--	FD0A	--	--
000B	--	--	FD0B	--	--
000C	--	--	FD0C	--	--
000D	--	--	FD0D	--	--
000E	--	--	FD0E	--	--
000F	--	--	FD0F	--	--

43. Turn the Trainer power off; then unplug the line cord. Wait twenty seconds; then plug in the line cord and turn on the Trainer.

44. Examine the memory locations listed in step 42, and write down the contents next to each address, in the second data column. Compare the two sets of data. Notice the data obtained at address 0000 through 000F changed when all Trainer power was removed. However, the data at address FD00 through FD0F is unchanged. Address 0000 is RAM, while address FD00 is ROM. Memory is lost from RAM when power is removed. When power is reapplied, random data will appear in the memory.

Enter FF at address FD00 through FD0F. Now examine address FD00 through FD0F. Notice the data is identical to that obtained in step 42. This shows that ROM can not be written into. You can send data down the data bus, but the memory will not accept it.

SUGGESTION: Use the nine instructions presented and write a few sample programs of your own. It's quite simple and can be great fun.

EXPERIMENT 4

Arithmetic and Logic Instructions

OBJECTIVES: *To present seven new instructions and use them in simple programs.*

To demonstrate 2's complement conversion.

To demonstrate binary subtraction.

To demonstrate binary addition of signed numbers.

To demonstrate logical manipulation of data using the AND and OR instructions.

Introduction

In Experiment 3, you used nine instructions to write various programs. These instructions were:

MNEMONIC	OP CODE	ADDRESSING MODE
LDA	86 ₁₆	Immediate
LDA	96 ₁₆	Direct
ADD	8B ₁₆	Immediate
ADD	9B ₁₆	Direct
STA	97 ₁₆	Direct
CLRA	4F ₁₆	Inherent
INCA	4C ₁₆	Inherent
DECA	4A ₁₆	Inherent
HLT	3E ₁₆	Inherent

Seven new instructions are presented in this experiment. Each is listed in Figure 7-14.

Unit 3 examined the process of binary arithmetic, 2's complement arithmetic, signed number addition, and Boolean logic. Through sample programs, this experiment will illustrate some of the operations presented in Unit 3.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Complement 2's or Negate (Inherent)	NEGA	0100 0000 ₂ or 40 ₁₆	Replace the contents of the accumulator with its complement plus 1.
Subtract (Immediate)	SUB	1000 0000 ₂ or 80 ₁₆	Subtract the contents of the next memory location from the contents of the accumulator. Place the difference in the accumulator.
Subtract (Direct)	SUB	1001 0000 ₂ or 90 ₁₆	Subtract the contents of the memory location whose address is given by the next byte from the present contents of the accumulator. Place the difference in the accumulator.
AND (Immediate)	ANDA	1000 0100 ₂ or 84 ₁₆	Perform the logical AND between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
AND (Direct)	ANDA	1001 0100 ₂ or 94 ₁₆	Perform the logical AND between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.
OR, Inclusive (Immediate)	ORA	1000 1010 ₂ or 8A ₁₆	Perform the logical OR between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
OR, Inclusive (Direct)	ORA	1001 1010 ₂ or 9A ₁₆	Perform the logical OR between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.

Figure 7-14
Instructions introduced in this experiment.

Procedure

1. In the first part of the experiment, you will determine how the microprocessor represents negative and positive numbers. The program shown in Figure 7-15 loads a positive number into the accumulator and then repeatedly decrements the number until it is negative. Enter this program into the Trainer. Verify that you entered it properly by examining each address.
2. Go to the single-step mode by: pressing the PC key; pressing the CHAN key; and entering the starting address (0000). Single-step through the program by repeatedly pressing the SS key. Notice that the first instruction places $+5_{10}$ in the accumulator. Refer to Figure 7-16 and record the contents of the accumulator (in both hexadecimal and binary) after each DECA instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate with 05.
0001	05	05	
0002	4A	DECA	Repeatedly decrement the accumulator.
0003	4A	DECA	
0004	4A	DECA	
0005	4A	DECA	
0006	4A	DECA	
0007	4A	DECA	
0008	4A	DECA	
0009	4A	DECA	
000A	4A	DECA	
000B	4A	DECA	
000C	4A	DECA	
000D	4A	DECA	
000E	4A	DECA	
000F	3E	HLT	Halt

Figure 7-15

This program decrements the contents of the accumulator from +5 to -8.

AFTER STEP	CONTENTS OF ACCUMULATOR		
	DECIMAL	HEXADECIMAL	BINARY
1	+5	05	0000 0101
2	+4		
3	+3		
4	+2		
5	+1		
6	0		
7	-1		
8	-2		
9	-3		
10	-4		
11	-5	FB	1111 1011

Figure 7-16
Record results here.

- In step 7, the number in the accumulator changed from 0 to -1. The microprocessor expresses -1 as $_{-16}$ or $_{-2}$. The table you have developed in Figure 7-16 shows how the microprocessor expresses the signed number from +5 to -5 in both hexadecimal and binary. The next program will add signed numbers like these.
- Enter the program shown in Figure 7-17. Use the single step mode to execute the program. What number is in the accumulator after the first instruction is executed? $_{-16}$ or $_{-2}$. What signed decimal number does this represent? _____.
- What number is in the accumulator after the second instruction is executed? $_{-16}$ or $_{-2}$. What decimal number does this represent? _____.
- What number is in the accumulator after the third instruction is executed? $_{-16}$ or $_{-2}$. What signed decimal number does this represent? _____.

Discussion

These very simple examples illustrate how the microprocessor represents signed numbers. Further experiments will show that the microprocessor can represent signed numbers between $+127_{10}$ and -128_{10} . You could determine the bit pattern for each negative number by clearing the accumulator and decrementing the required number of times. However, there are much simpler ways of determining the proper bit pattern for negative numbers.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	+5	with +5.
0002	8B	ADD	Add immediate
0003	FB	-5	-5.
0004	8B	ADD	Add immediate
0005	FC	-4	-4
0006	3E	HLT	

Figure 7-17
Adding signed numbers.

The simplest way is to start with the positive binary equivalent and take the two's complement by changing all 0's to 1's and 1's to 0's and adding 1. The microprocessor has an instruction that will do this for us. It is called the two's complement or Negate instruction. Its mnemonic is NEGA. This instruction changes the number in the accumulator to its two's complement. It is used to change the sign of a number.

Procedure (continued)

7. Load the program shown in Figure 7-18. Use the single-step mode to execute the program. Execute the first instruction by depressing the SS key. What number is in the accumulator? $_{-16}$ or $_{-2}$. What signed decimal number does this represent? _____.
8. Execute the second instruction. What number is in the accumulator? $_{-16}$ or $_{-2}$. What signed decimal number does this represent? _____. Compare this with the number in step 7. What affect did the NEGA instruction have? _____.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	+5	with +5.
0002	40	NEGA	Change the number to -5.
0003	40	NEGA	Change it back to +5.
0004	4A	DECA	Decrement the number to +4.
0005	40	NEGA	Change the number to -4.
0006	40	NEGA	Change it back to +4.
0007	3E	HLT	Halt

Figure 7-18
Using the NEGA instruction.

9. Execute the third instruction. What number is in the accumulator?
 $-_{16}$ or $-----_2$. What signed decimal number does this represent? _____. Is your answer the same as that found in step 7? _____.
10. Execute the fourth instruction. This decrements the accumulator so that it now contains the signed decimal number _____.
11. Execute the fifth instruction. What number is in the accumulator?
 $-_{16}$ or $-----_2$. What signed decimal number does this represent? _____.
12. Execute the sixth instruction. The number in the accumulator is
 $-_{16}$ once more.

Discussion

The program used the NEGA instruction four times. The first time, the NEGA instruction changed 05_{16} to its two's complement FB_{16} . Referring back to the table you developed in Figure 7-16, this is the representation for -5_{10} . Thus, the NEGA instruction effectively changes the sign of the number in the accumulator. The next step proved this again by converting -5_{10} back to $+5_{10}$. To further emphasize the point, the number was decremented to $+4_{10}$. The next NEGA instruction changed this to FC_{16} which is the representation for -4_{10} . The final NEGA instruction converts this back to $+4_{10}$. This instruction allows us to convert a positive number to its negative equivalent and vice versa.

In Unit 3, you learned that the MPU can work with signed numbers in the range of $+127_{10}$ to -128_{10} or unsigned numbers in the range of 0 to 255_{10} . This capability results from the way we interpret bit patterns. The following steps will demonstrate this.

Procedure (continued)

13. Figure 7-19 shows a program for adding the unsigned numbers 220_{10} and 27_{10} . Load this program into the Trainer and execute it. The final result in the accumulator is $_{16}$ or $-----_2$. What unsigned decimal number does this represent? _____.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	DC	220 ₁₀	with 220 ₁₀ .
0002	8B	ADD	Add immediate
0003	1B	27 ₁₀	27 ₁₀ .
0004	3E	HLT	Halt.

Figure 7-19
Adding unsigned numbers.

14. Figure 7-20 shows a program for adding the signed numbers -36_{10} and 27_{10} . Load and execute this program. The final result in the accumulator is $_{16}$ or $_{2}$. What signed decimal number does this represent? _____.
15. Compare the results obtained in steps 13 and 14. Compare the HEX Contents columns of Figure 7-19 with that of Figure 7-20.

Discussion

This demonstrates that the MPU simply adds bit patterns. It is our interpretation of these patterns that decide whether we are using signed or unsigned numbers. After all, the two programs are identical except for our interpretation of the input and output data.

Negative numbers are often encountered when performing subtract operations. The subtract instruction was shown earlier in Figure 7-14. Either immediate or direct addressing can be used.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	DC	-36_{10}	with -36_{10}
0002	8B	ADD	Add immediate
0003	1B	$+27_{10}$	$+27_{10}$
0004	3E	HLT	Halt.

Figure 7-20
Adding signed numbers.

Procedure (continued)

16. Load the program shown in Figure 7-21. Execute the program using the single-step mode. What is the number in the accumulator after the first subtract instruction is executed? $_{16}$ or $_{2}$ or $_{10}$.
17. What is the number in the accumulator after the second subtract instruction is executed? $_{16}$ or $_{2}$. What signed decimal number does this represent? _____.

Discussion

The first subtract instruction subtracted 16_{10} from 47_{10} , leaving 31_{10} . The second one subtracted 35_{10} from 31_{10} . This produced a result of -4_{10} . However, the MPU expressed -4 in two's complement form (FC_{16} or $1111\ 1100_2$). You will find this to be the case anytime the MPU produces a negative result.

Now let's look at some of the logical instructions available to the microprocessor. The AND and OR instructions are described in Figure 7-14. Carefully read the description of these instructions given there. While these instructions have many uses, we will demonstrate only one here. Earlier you learned that certain peripheral devices communicate with computers using the ASCII code. Thus, when the "2" key on a teletypewriter is pushed, the computer receives the ASCII code for 2, which is $0011\ 0010$. The ASCII code for 6 is $0011\ 0110$. Notice that the four least significant bits of the ASCII character are the binary value of the corresponding numeral. Thus, we can convert the ASCII characters for the numerals 0 through 9 to binary simply by setting the four most significant bits to 0's. Likewise, we can convert the binary numbers $0000\ 0000$ through $0000\ 1001$ to ASCII by changing the four most significant bits to 0011 .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	2F	47_{10}	with 47_{10} .
0002	80	SUB	Subtract immediate
0003	10	16_{10}	16_{10}
0004	80	SUB	Subtract immediate
0005	23	35_{10}	35_{10}
0006	3E	HLT	Halt

Figure 7-21
Using the subtract instruction

Procedure (continued)

18. Load the program shown in Figure 7-22. Single-step through the first instruction. The number in the accumulator is _____₂.
19. Execute the second instruction. This AND's the contents of the accumulator with the "mask" _____. The number in the accumulator after this AND operation is _____₂. Compare this with the number that was in the accumulator in step 18. Compare both numbers with the mask. A 1 in the original number is retained only if there is a _____ in the corresponding bit position of the mask.
20. Execute the third instruction. In what memory location is the number in the accumulator stored? _____₁₆. What number is now in the accumulator? _____₂. Does the number still appear in the accumulator after being stored in memory? _____.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator with
0001	0B	OB	the ASCII character at this address.
0002	84	AND	AND it with
0003	0F	OF	this "mask".
0004	97	STA	Store the binary equivalent
0005	0C	OC	at this address.
0006	8A	ORA	OR the number with
0007	30	30	this "mask".
0008	97	STA	Store the result
0009	0D	OD	here.
000A	3E	HLT	Stop
000B	37	0011 0111	ASCII character for numeral 7.
000C	—	—	Reserved
000D	—	—	Reserved

Figure 7-22
Using the AND and OR instruction.

21. Execute the fourth instruction. This OR's the contents of the accumulator with the "mask" _ _ _ _ _₂. The number in the accumulator is _ _ _ _ _₂. Compare this with the mask and the number that was in the accumulator in step 20. A 1 is produced in the result whenever there is a _____ in the corresponding bit position of either the original number, the mask, or both.
22. Execute the fifth instruction. This stores the number in memory location _____₁₆.
23. Examine memory locations 000B₁₆, 000C₁₆, and 000D₁₆ and compare their contents.

Discussion

The program first converts the ASCII code for the number "7" to the binary number 0000 0111. It does this by ANDing the ASCII code with the "mask" 0000 1111₂. Notice that a 1 bit in the mask allows the corresponding bit in the original number to be retained. The four most significant bits of the original number are "masked off" because they are ANDed with 0's.

The OR operation restores the ASCII character by attaching 0011 as the four most significant bits.

EXPERIMENT 5

Program Branches

OBJECTIVES:

To manipulate the N, Z, V, and C condition code registers and determine the conditions that set and reset these flags.

To verify the operation of a simple multiply by repeated addition program that uses the BEQ conditional branch instruction and the BRA instruction.

To demonstrate the ability to write a program that divides by repeated subtraction and uses a conditional branch and BRA instruction.

To introduce a shorthand method of calculating relative addresses.

To verify the operation of a program that converts BCD numbers to their binary equivalent.

To demonstrate the effect an incorrect relative address can have on a program operation and how the microprocessor trainer can be used to debug programs.

Introduction

As mentioned previously, conditional branch instructions give the computer the power to make decisions. As the name implies, a certain condition must be met before a branch takes place. The condition code registers monitor the accumulator and signal the presence of a specific condition. If the MPU encounters a conditional branch instruction, it merely checks the condition code registers, or flags, to see if the condition is satisfied. If the specific flag is set, the program branches off to another section. If not, the normal program continues.

Therefore, the conditional branch instructions inherit their power from these simple condition code registers. A sound knowledge of how these flags are set and cleared will enhance your ability as a programmer.

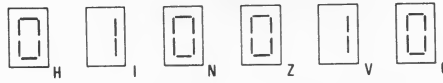


Figure 9-23

Displaying the conditions of the flags.

Since condition code registers are very important, your Trainer was designed with a special key to allow you to examine these flags. The key is labelled “CC” for “Condition Code.” When this key is pressed, the state of the condition code registers will be displayed. Each LED displays the contents of one register. The letter just to the right of each LED denotes the corresponding register as shown in Figure 7-23.

Notice that there are six flag registers. For the moment we aren’t concerned with the two left-most flags. They will be covered in a later unit. However, we are interested in the N, Z, V, and C flags, because they indicate conditions that can lead to conditional branches. Notice that the flags can either be set as indicated by a 1 or they can be cleared as indicated by a 0.

In this first portion of the experiment, you will implement a “do-nothing” program that manipulates the condition code registers. Then single-stepping through the program, you will examine how the accumulator changes these flags.

Procedure

1. Turn on the Trainer and then press the RESET key.
2. Now, load the program listed in Figure 7-24 into the Trainer. Once the program is loaded, go back and examine it to insure that it’s entered correctly.

Now look at the first instruction of the program in Figure 7-24. It has the op code 01 and the mnemonic is “NOP.” As the comments column points out, this is a “do-nothing” type of instruction called a “No-Op.” In other words, it performs no operation. In this program, the NOP’s primary function is to allow you to see the first instruction before it’s executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	"DO Nothing" Instruction
0001	86	LDA	Load the accumulator immediate
0002	FF	FF ₁₆	with FF ₁₆ .
0003	86	LDA	Load the accumulator immediate
0004	77	77 ₁₆	with 77 ₁₆ .
0005	86	LDA	Load the accumulator immediate
0006	00	00 ₁₆	with 00 ₁₆ .
0007	86	LDA	Load the accumulator immediate
0008	01	01 ₁₆	with 01 ₁₆ .
0009	86	LDA	Load the accumulator immediate
000A	92	92 ₁₆	with 92 ₁₆ .
000B	8B	ADD	Add Immediate
000C	C6	C6 ₁₆	C6 ₁₆ .
000D	86	LDA	Load the accumulator immediate
000E	08	08 ₁₆	with 08 ₁₆ .
000F	8B	ADD	Add Immediate
0010	08	08 ₁₆	08 ₁₆ .
0011	86	LDA	Load the accumulator immediate
0012	01	01 ₁₆	with 01 ₁₆ .
0013	80	SUB	Subtract immediate
0014	02	02 ₁₆	02 ₁₆ .
0015	86	LDA	Load the accumulator immediate
0016	77	77 ₁₆	with 77 ₁₆ .
0017	80	SUB	Subtract immediate
0018	66	66 ₁₆	66 ₁₆ .
0019	86	LDA	Load the accumulator immediate
001A	49	49 ₁₆	with 49 ₁₆ .
001B	8B	ADD	Add immediate
001C	60	60 ₁₆	60 ₁₆ .
001D	86	LDA	Load the accumulator immediate
001E	10	10 ₁₆	with 10 ₁₆ .
001F	3E	HLT	Halt.

Figure 7-24

Program to illustrate the condition code registers.

In previous experiments, you probably noticed that when you single-stepped through programs, you never saw the first instruction. This is because in the "SS" mode, the Trainer executes the first instruction automatically and then stops on the second instruction. This can be somewhat confusing.

To offset this problem, we merely insert the NOP. The Trainer "sees" this as the first instruction, although nothing is accomplished by the NOP. Therefore, the Trainer displays the next instruction, which is the first "real" instruction of the program, permitting you to view it before it's executed.

3. Load the program counter with address 0000 and then press the SS key. Recall that the first four displays represent the address that's currently in the program counter. The two right-most displays show the op code stored at this address. Record the information below.

PC _ _ _ _ 1 OP CODE 86 _ _

Now, press the ACCA key and record the contents of the accumulator.

ACCA 20 _ _

The contents of the accumulator will be a random number, since we haven't yet executed a program instruction.

Now, press the CC key and record the contents of the N, Z, V, and C condition code registers below.

0110
_ _ _ _

N Z V C

Again, the states of the flags are random at this time.

4. Now, press the SS key and then the ACCA key. Record the contents of the accumulator below.

ACCA FF _ _

Press key CC and record the state of the N flag below.

1 _ _ _
N Z V C

With the negative number FF_{16} in the accumulator, the negative (N) flag is set.

5. Press the SS key again. The program count should now be 0005_{16} and the op code at this address is 86. Now check and record the contents of the accumulator and the N flag.

ACCA 77 0 - - -
N Z V C

With the positive number 77_{16} in the accumulator, the N flag is cleared, or reset, to 0.

From the information gathered in steps 4 and 5, what conclusions do you reach with respect to the N flag and the contents of the accumulator?

6. Single-step the program again. The program count is now 0007_{16} . Record the contents of the accumulator and the condition of the Z flag below.

ACCA 00 1 - - -
N Z V C

With 00_{16} in the accumulator, the Z flag is set.

Press SS and again record the contents of the accumulator and the Z flag below.

ACCA 01 0 - - -
N Z V C

The accumulator now contains 01_{16} and the Z flag is cleared. What is the relation between the contents of the accumulator and the Z, or zero flag?

7. Single-step again and record the information below.

ACCA 58 00 = -
N Z V C

This step loads the number 92_{16} into the accumulator. Bit 7 of the accumulator contains a 1_2 so the N flag is set. Naturally, the Z flag is cleared. The next instruction will add $C6_{16}$ to the contents of the accumulator. As shown below, this operation should generate a carry.

1001	0010	=	92_{16}
1100	0110	=	$C6_{16}$
<hr/>			
0101	1000	=	158_{16}

CARRY $\xrightarrow{1}$

- Press the SS key and record the information below.

ACCA 08 00 = 1
N Z V C

The 8-bit accumulator cannot hold the 9-bit sum. However, the carry generated by the addition sets the C flag.

8. This step loads the number 08_{16} into the accumulator. Press the SS key and record the information below.

ACCA 18 00 = 00
N Z V C

Notice that loading this new number into the accumulator didn't affect the carry (C) flag. The next step will add 08_{16} to the contents of the accumulator (08_{16}).

9. Press the SS key and record the information below.

ACCA 01 00 = 00
N Z V C

The accumulator now contains the sum of the addition (10_{16}) and the carry flag is cleared.

From the results of steps 8 and 9, you might conclude that the carry flag can be cleared by another _____ that does not result in a carry.

10. Press the SS key. The program count should now be 0013. Record the information below.

ACCA _ _ _ _
N Z V C

This shows that the accumulator contains 01_{16} and that the N, Z, and C flags are all cleared. When the next instruction is executed, the number 02_{16} will be subtracted from 01_{16} (the contents of the accumulator). As shown below, the subtraction should result in a borrow, setting the C flag.

Borrow		0000	0001	=	01_{16}
		0000	0010	=	02_{16}
		<hr/>			
		1111	1111	=	FF_{16}

Notice that the difference is FF_{16} . This will _____ the N flag.
set/clear

11. Press the SS key and record the information below.

ACCA _ _ _ _
N Z V C

As expected, the difference produced is FF_{16} . Also, the N flag is set, indicating a negative number is in the accumulator and the C flag indicates a borrow occurred.

The next step will execute the instruction that loads 77_{16} into the accumulator. After this LDA operation, the C flag will be _____.
set/cleared

12. Press the SS key and record the information below.

ACCA _ _ _ _ = _
N Z V C

Notice that the C flag is still set and that 77_{16} is in the accumulator. Now we will subtract 66_{16} from the accumulator contents (77_{16}).

Press the SS key and record the information below.

ACCA _ _ _ _ = _
N Z V C

The difference (11_{16}) is now stored in the accumulator and, since no borrow is generated, the C flag is cleared.

13. In this step, the first instruction loads the accumulator with the number 49_{16} . The next instruction adds the number 60_{16} to 49_{16} . As shown below, the addition of these numbers causes an overflow into the sign bit (bit 7) and the sum, $A9_{16}$, appears to be a negative number.

0100	1001	=	49_{16}
0110	0000	=	60_{16}
<hr/>			
1010	1001	=	$A9_{16}$

Overflow changes —
sign bit.

Of course, this is incorrect and the MPU must be notified of this overflow. This is the purpose of the V flag.

Press the SS key and record the information below.

ACCA _ _ _ _ _ _
N Z V C

The number 49_{16} is in the accumulator and the N, Z, V, and C flags are cleared.

Single-step once more and then record the information below.

```

ACCA --  -- -- --
        N Z V C

```

The sum $A9_{16}$ is now in the accumulator. Notice that the N and V flags are set, indicating that the number in the accumulator is negative and that an overflow occurred.

14. When the next instruction is executed, the number 10_{16} will be loaded into the accumulator.

Single-step the program and record the information below. Notice that the op code 3E (a halt) is the next instruction, so the program is finished.

```

ACCA --  -- -- --
        N Z V C

```

The accumulator contains the number 10_{16} , and all flags cleared. From this, you might conclude that any instruction that doesn't produce an overflow in the accumulator will _____ the V flag.

set/clear

Discussion

In this portion of the experiment, you stepped through a simple program that manipulated the condition code registers. In step 4, the negative number FF_{16} was loaded into the accumulator. This set the N flag to 1₂. In step 5, the positive number 77_{16} was loaded into the accumulator. And, as you noted, the N flag was cleared or reset to 0₂. From these two steps you should have concluded that when the number in the accumulator is negative, the N flag is set. And when the accumulator contains a positive number, the N flag is cleared.

In step 6, the accumulator was loaded with 00_{16} . This set the Z flag to 1₂. Next, when 01_{16} was loaded, the Z flag was reset or cleared to 0₂. Your conclusion should have been that when the accumulator contains 00_{16} , the Z flag is set. If it contains any number other than 00_{16} , the Z flag is cleared.

Next, you examined the C flag. When a carry was generated by the addition of the two numbers, 92_{16} and $C6_{16}$, the C flag was set. In step 8, you noted that merely loading a new number into the accumulator did not clear the C flag. The carry flag was cleared by another addition that did not result in a carry. Your conclusion should have been that the C flag can only be cleared by an arithmetic operation that does not result in a carry.

As you proved in steps 10 and 11, a subtraction that results in a borrow also sets the C flag. Again, the C flag was cleared by an arithmetic operation, in this case a subtraction, that did not generate a borrow. Therefore, the C flag can only be cleared or reset to 0_2 by an arithmetic operation that does not result in a borrow or carry.

You concluded this phase of the experiment by adding two positive numbers, the sum of which overflowed into the sign bit of the accumulator. This set the V or overflow flag, showing that the sum should not be a negative number as the N flag indicated. The next LDA instruction cleared the V flag. From this, you should conclude that the V flag is cleared by any instruction that doesn't produce an overflow.

In the next sections of this experiment, you will step through a few branching programs that illustrate the use of the branch always (BRA) instruction and certain conditional branch instructions. These branch instructions were discussed in Unit 4, and you will verify their operation. We'll begin with the multiply by repeated addition program.

Procedure (continued)

15. Enter the program listed in Figure 7-25 into the Trainer. This program multiplies 05_{16} and 02_{16} and stores the product in address 0013_{16} . Recheck the program to insure that it's entered correctly.
16. This is the same program that you stepped through in Unit 4. Notice that the program contains two branch instructions; the BEQ (Branch if Equal Zero) at address 0005_{16} and the BRA (Branch Always) at address $000E_{16}$.

The branch if equal zero (BEQ) instruction implies by it's name that a conditional branch will occur when the _____ flag is set.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	→ STA	Store the product
0002	13	13	in location 13_{16} .
0003	96	LDA	Load the accumulator with the
0004	12	12	multiplier from location 12_{16} .
0005	27	BEQ	If the multiplier is equal to zero,
0006	09	09	branch down to the Halt instruction.
0007	4A	DECA	Otherwise, decrement the multiplier.
0008	97	STA	Store the new value of the
0009	12	12	multiplier back in location 12_{16} .
000A	96	LDA	Load the accumulator with the
000B	13	13	product from location 13_{16} .
000C	9B	ADD	Add
000D	11	11	the multiplicand to the product.
000E	20	BRA	Branch back to instruction
000F	F1	F1	in location 01.
0010	3E	→ HLT	Halt.
0011	05	05	Multiplicand.
0012	02	02	Multiplier.
0013	—	—	Product.

Figure 7-25
Program to multiply by repeated addition.

17. Now, set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 9-26. Notice that you will be monitoring the Z flag. A comments column is provided so you can make notes about each step. Use the program listing as a reference for each op code and the corresponding operand.

18. When the BEQ instruction is executed and the Z flag is set, the program branches to the _____ instruction.

When the multiplier was 02_{16} , the program halted on the _____ pass through the program.

If the multiplier is changed to 06_{16} , how many passes would the program make before it halts? _____.

19. Examine the contents of address 0013_{16} and record below.

0013 _ _.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure 7-26

Single-stepping through the Multiply by repeated addition program.

Discussion

The chart that you completed should be similar to the one shown in Figure 7-27. Compare the charts.

The first step we don't see, since it's executed before the Trainer stops at address 0001. Nevertheless, we do see the result of this clear accumulator instruction because the accumulator contains 00. When step 1 is executed, 00₁₆ is stored in location 0013₁₆. Step 2 brings us to address 0003₁₆ which loads the accumulator with the multiplier, in this example, 02₁₆. The BEQ instruction is next, but the Z flag is cleared so the program continues on the normal route. Next the multiplier is decremented to 01₁₆ and then stored in location 0012₁₆. Now the product (00₁₆) is loaded and the multiplicand (05₁₆) is added directly. This produces the new product, 05₁₆. Now the program encounters the BRA, or branch always instruction and it branches back to address 0001₁₆.

Here the new product is stored away and the multiplier is loaded again. It's 01₁₆ this time, so the program continues on through the BEQ instruction, the multiplier is decremented to 00₁₆, and the multiplicand 05₁₆ is added to the product. The new product (0A₁₆) is still in the accumulator. Once again, the BRA instruction loops flow back to address 0001₁₆ and the product is stored in address 0013₁₆.

The multiplier is now loaded and, since it's been decremented to 00₁₆, it sets the Z flag. The BEQ instruction checks the Z flag, finds that it's set and branches to the halt instruction at address 0010₁₆. Therefore, the program makes two complete passes, before the multiplier becomes 00₁₆. On the third pass through, BEQ terminates the program because the Z flag is set.

The multiplier sets the count and determines how many additions will be performed. If the multiplier is changed to 06₁₆, the program will make six complete loops, halting on the seventh loop. The BEQ will only be satisfied when the multiplier has been reduced to 00.

All branch instructions use relative addressing. In Unit 4, we discussed the method used to calculate the destination address for a branch instruction. However, another shorthand type procedure that's quite popular with programmers can be used. With this technique, you simply count in hexadecimal. For a forward branch, you begin at 00₁₆ and count up to the destination address.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0001	97	00	1	Store the product (00 ₁₆) in address 0013 ₁₆ .
2	0003	96	00	1	Load the accumulator with the multiplier (02 ₁₆) from address 0012 ₁₆ .
3	0005	27	02 ↑ Multiplier	0	BEQ. Check the Z flag. It's not set so continue.
4	0007	4A	02	0	Decrement the multiplier (02 ₁₆).
5	0008	97	01 ↑ New Multiplier	0	Store the new multiplier (01 ₁₆) at address 0012 ₁₆ .
6	000A	96	01	0	Load the accumulator with the product (00) at address 0013 ₁₆ .
7	000C	9B	00	1	Add the multiplicand (05) giving new product.
8	000E	20	05 ↑ New Product	0	Branch back to address 0001 ₁₆ .
9	0001	97	05	0	Store the product (05 ₁₆) in address 0013 ₁₆ .
10	0003	96	05	0	Load the accumulator with the multiplier (01 ₁₆) located at address 0012 ₁₆ .
11	0005	27	01	0	BEQ. Check Z flag. It's not set so continue.
12	0007	4A	01	0	Decrement the multiplier (01 ₁₆).
13	0008	97	00 ↑ New Multiplier	1	Store the new Multiplier (00 ₁₆) at address 0012 ₁₆ .
14	000A	96	00	1	Load the accumulator with the product (05 ₁₆) at address 0013 ₁₆ .
15	000C	9B	05	0	Add the multiplicand (05 ₁₆) giving new product.
16	000E	20	0A ↑ New Product	0	Branch back to address 0001 ₁₆ .
17	0001	97	0A	0	Store the product (0A ₁₆) in address 0013 ₁₆ .
18	0003	96	0A	0	Load the accumulator with the multiplier (00 ₁₆) from address 0012 ₁₆ .
19	0005	27	00	1	BEQ. Check the Z flag. Now it's set. Branch to address 0010 ₁₆ .
20	0010	3E	00	1	Halt.

Figure 7-27

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	??	??
1A		
1B		
1C		
1D		
1E		
1F		
20		
21		
22		
23		
24		

Originating address

Destination address

We wish to
Branch to here

Figure 7-28

For example, in the program of Figure 9-28, we want to branch from address 18_{16} to address 24_{16} . Recall that the relative address is added to the contents of the program counter. After the BRA instruction and its operand (the relative address) have been fetched, the program counter is pointing to address $1A_{16}$. Therefore, we begin our count at address $1A_{16}$. Then we count forward in hex as shown in Figure 9-29. When we reach the destination address, the hexadecimal count is the relative address. In this case, it's $0A_{16}$, and we insert this operand at address 19_{16} .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	0A	0A
00 1A		
01 1B		
02 1C		
03 1D		
04 1E		
05 1F		
06 20		
07 21		
08 22		
09 23		
0A 24		

Originating Address

Destination Address

Relative
Address

Figure 7-29
Branching forward

To branch backward in the program, we simply count down using negative hex numbers. It may sound more difficult, but once you are accustomed to it, you will find it easier to use than the previous method you learned.

For example, in the program shown in Figure 7-30A, we wish to branch back to address 58_{16} . The BRA instruction, at address $5D_{16}$ is fetched and the program count points to address $5F_{16}$. Figure 7-30B shows how we calculate the address for this backward branch. We begin with FF_{16} , and count down. When we reach the destination address (58_{16}), the count at that point is the relative address, in this case $F9_{16}$.

Figure 7-31 shows another example of computing the relative address for a larger branch. The branch instruction is at address $B0_{16}$ and therefore, the origination address is $B2_{16}$. We calculate the relative address as shown in Figure 7-31B. Starting with FF_{16} at address $B1_{16}$ we count down to the destination address $A0_{16}$. As the count indicates, the relative address to get to $A0_{16}$ is EE_{16} .

A	HEX ADDRESS			HEX CONTENTS		MNEMONICS/ HEX CONTENTS	
	56			—		—	
	57			—		—	
	58	← Destination		—		—	
	59	Address		—		—	
	5A			—		—	
	5B			—		—	
	5C			—		—	
	5D			20		BRA	
	5E	← Originating		??		??	
	5F	Address					
Program branches to here							

B	HEX ADDRESS			HEX CONTENTS		MNEMONICS/ HEX CONTENTS	
	56			—		—	
	57			—		—	
	F9	← Destination		—		—	
	FA	Address		—		—	
	FE	5A		—		—	
	FC	5B		—		—	
	FD	5C		—		—	
	FE	5D		20		BRA	
	FF	5E	← Originating	F9		F9	
		Address					
	5F						
Relative address							

Figure 7-30
Branching back

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
AO	—	—
A1	—	—
A2	—	—
A3	—	—
A4	—	—
A5	—	—
A6	—	—
A7	—	—
A8	—	—
A9	—	—
AA	—	—
AB	—	—
AC	—	—
AD	—	—
AE	—	—
AF	—	—
BO	26	BNE
B1	??	??
B2	—	—

We wish to branch to here

Destination Address

Originating Address

A

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
EE AO	—	—
EF A1	—	—
FO A2	—	—
F1 A3	—	—
F2 A4	—	—
F3 A5	—	—
F4 A6	—	—
F5 A7	—	—
F6 A8	—	—
F7 A9	—	—
F8 AA	—	—
F9 AB	—	—
FA AC	—	—
FB AD	—	—
FC AE	—	—
FD AF	—	—
FE BO	26	BNE
FF B1	EE	EE
B2	—	—

Relative Address

B

Figure 7-31

In the next section of this experiment, you will write a program that will divide by repeated subtraction. You will probably have two branches in this program; a forward branch and a branch back. Use this new technique to calculate the relative addresses for both branches.

Procedure (continued)

20. In Unit 4, we discussed a program that divides by repeated subtraction. The flow chart for this program is shown in Figure 7-32. Using this flow chart as a guide and the instructions presented in Figure 7-33, write a program that divides by repeated subtraction.

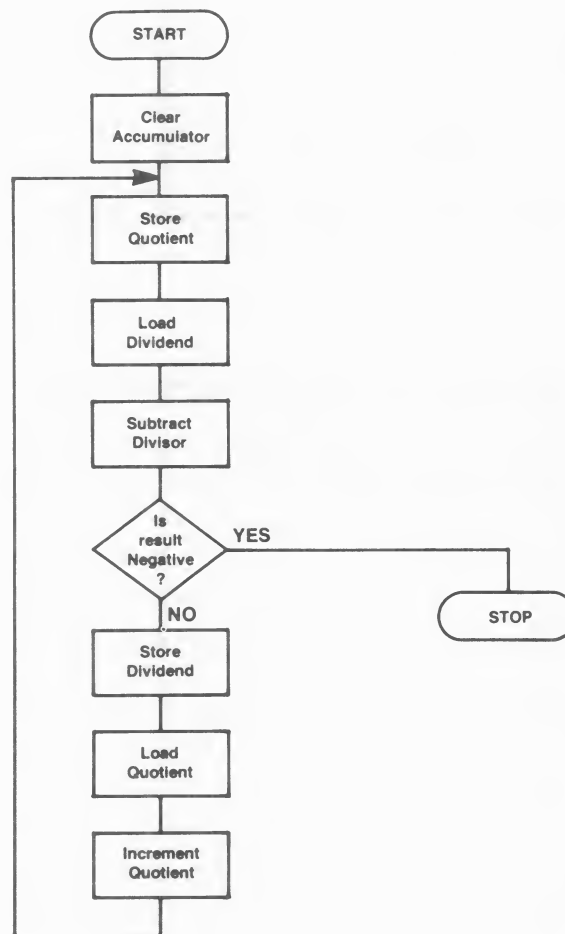


Figure 7-32

Flow chart for dividing by repeated subtraction.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch if Carry Set	BCS			25	
Branch if	BEQ			27	
Equal Zero					
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 7-33
Instructions to be used.

21. Now load the program into the Trainer. Let the dividend be $0B_{16}$ and the divisor be 05_{16} . Change the program counter to the starting address of your program and single-step through the program, recording the information in the chart of Figure 9-34.
22. Examine the contents of the address that stores the dividend and the quotient. If you followed the flow chart, the address where the dividend is stored should now contain the remainder from the division. Record the contents below.

Quotient _____ Remainder _____

Discussion (continued)

Now you've written a program that incorporates an unconditional branch and a conditional branch. Hopefully, you calculated the relative addresses using the shorthand technique just discussed. Our program for the divide by repeated subtraction is listed in Figure 9-35. If you followed the flow chart, your program should be similar to this.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	→ STA	Store in the quotient which
0002	13	13	is at address location 13 ₁₆ .
0003	96	LDA	Load the accumulator with the
0004	11	11	dividend from location 11 ₁₆ .
0005	90	SUB	Subtract the
0006	12	12	divisor from the dividend.
0007	2B	BMI	If the difference is negative,
0008	07	07	branch down to the Halt
			instruction.
0009	97	STA	Otherwise, store the difference
000A	11	11	back in location 11 ₁₆ .
000B	96	LDA	Load the accumulator with the
000C	13	13	quotient.
000D	4C	INCA	Increment the quotient by one.
000E	20	BRA	Branch back to instruction
000F	F1	F1	in location 01.
0010	3E	→ HLT	Halt.
0011	0B	0B	Dividend (11 ₁₆).
0012	05	05	Divisor (5 ₁₆).
0013	—	—	Quotient.

Figure 7-35
Dividing by repeated subtraction.

STEP	PROGRAM COUNTER	OPCODE	ACCA	N FLAG	COMMENTS
1	0001	97	00	0	Store the quotient (00 ₁₆) at address 0013 ₁₆ .
2	0003	96	00	0	Load the accumulator with the dividend from address 0011 ₁₆ .
3	0005	90	OB ↑ Dividend	0	Subtract the divisor (05 ₁₆) at address 0012 ₁₆ from the accumulator.
4	0007	2B	06 ↑ After subtraction	0	BMI. Check the N flag. It's not set so continue.
5	0009	97	06	0	Store the difference (06 ₁₆) back in address 0011 ₁₆ .
6	000B	96	06	0	Load the accumulator with the quotient (00 ₁₆) at address 0013 ₁₆ .
7	000D	4C	00	0	Increment the quotient.
8	000E	20	01 ↑ Quotient after INC	0	Branch back to the instruction at address 0001 ₁₆ .
9	0001	97	01	0	Store the quotient (01 ₁₆) at address 0013 ₁₆ .
10	0003	96	01	0	Load the accumulator with the dividend (06 ₁₆) at address 0011 ₁₆ .
11	0005	90	06 ↑ Dividend Now	0	Subtract the divisor (05 ₁₆) at address 0012 ₁₆ from the accumulator.
12	0007	2B	01 ↑ After Subtraction	0	BMI. Check the N flag. It's not set so continue.
13	0009	97	01	0	Store the difference (01 ₁₆) back in address 0011 ₁₆ .
14	000B	96	01	0	Load the accumulator with the quotient (01 ₁₆) at address 0013 ₁₆ .
15	000D	4C	01	0	Increment the quotient.
16	000E	20	02 ↑ Quotient after INC.	0	Branch back to the instruction at address 0001 ₁₆ .
17	0001	97	02	0	Store the quotient (02 ₁₆) at address 0013 ₁₆ .
18	0002	96	02	0	Load the accumulator with the dividend (01 ₁₆) at address 0011 ₁₆ .
19	0005	90	01	0	Subtract the divisor (05 ₁₆) at address 0012 ₁₆ from the accumulator.
20	0007	2B	FC ↑ Negative Number	1	BMI. Check the N flag. Now it's set so branch to the instruction at address 0010 ₁₆ .
21	0010	3E	FC	1	Halt.

Figure 7-36

Notice that we used the BMI (Branch if Minus) conditional branch instruction. Therefore, the N or negative flag will satisfy the branch when it's set. Figure 7-36 charts our program as we single-stepped through it. Since the program subtracts the divisor from the dividend and stores the difference as the new dividend, at the conclusion of the program the dividend is actually the remainder of the division. When the $0B_{16}$ is divided by 05_{16} , the quotient should be 02_{16} and the remainder 01_{16} .

So far, we've used the conditional branch instructions only to exit a loop and then halt program execution. However, these branch instructions become even more powerful when they are used to "chain" together different portions of a program. Figure 7-37 shows an example of this chaining effect. The program starts and runs through the first loop until the conditional branch BEQ is satisfied. Then it exits this loop and starts another. When the BEQ condition is satisfied in the second loop, another exit is performed, and another portion of the program is executed.

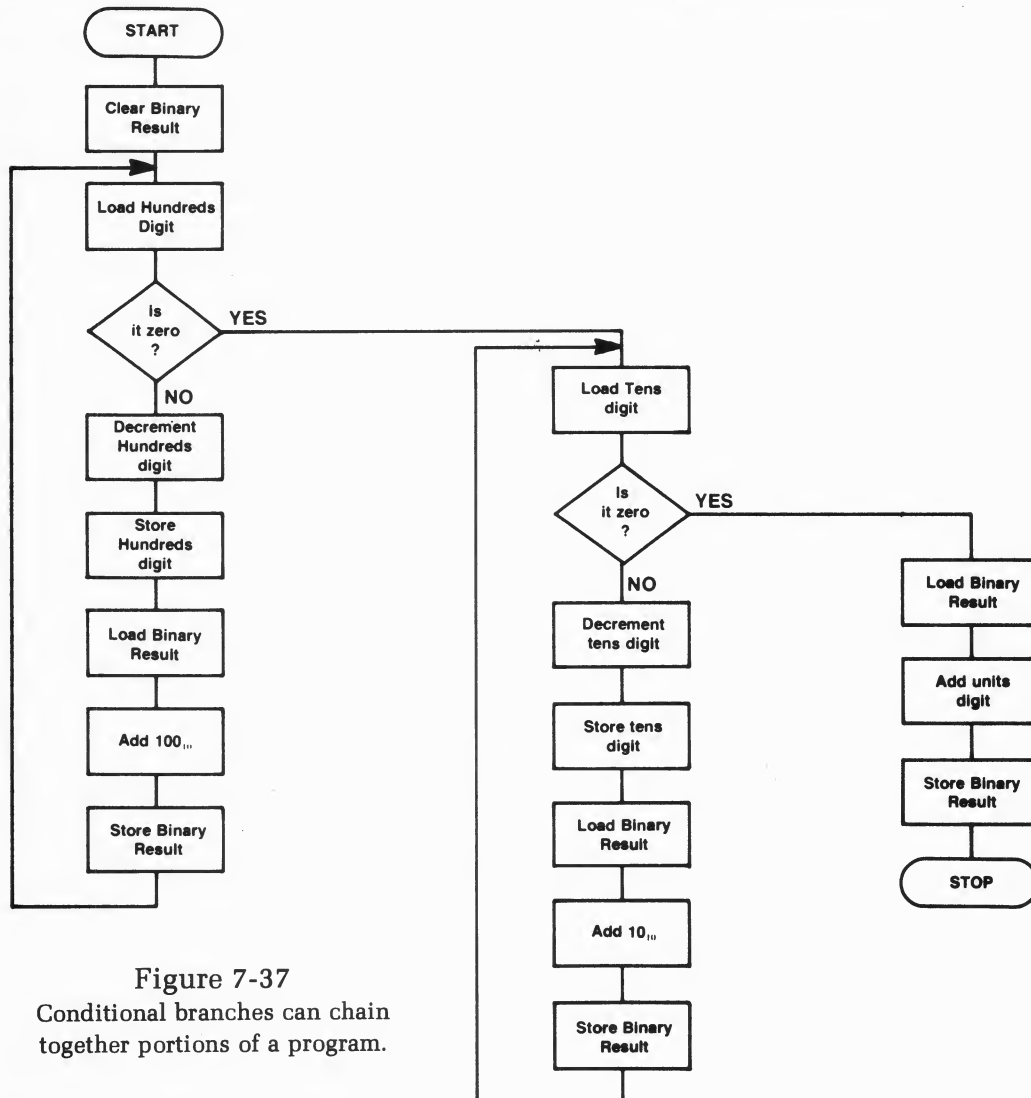


Figure 7-37
Conditional branches can chain together portions of a program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the Accumulator.
0001	97	STA	Store 00
0002	2B	2B	in location 2B. This clears the binary result.
0003	96	LDA	Load direct into the accumulator
0004	28	28	the hundreds BCD digit.
0005	27	BEQ	If the hundreds digit is zero, branch
0006	OB	OB	forward to the instruction in location 12 ₁₆ .
0007	4A	DECA	Otherwise, decrement the accumulator.
0008	97	STA	Store the result as the new
0009	28	28	hundreds BCD digit.
000A	96	LDA	Load direct into the accumulator
000B	2B	2B	the binary result.
000C	8B	ADD	Add immediate
000D	64	64	100 ₁₀ to the binary result.
000E	97	STA	Store away the new
000F	2B	2B	binary result.
0010	20	BRA	Branch
0011	F1	F1	back to the instruction in location 03 ₁₆ .
0012	96	LDA	Load direct into the accumulator
0013	29	29	the tens BCD digit.
0014	27	BEQ	If the tens BCD digit is zero, branch
0015	OB	OB	forward to the instruction in location 21 ₁₆ .
0016	4A	DECA	Otherwise, decrement the accumulator.
0017	97	STA	Store the result as the new
0018	29	29	tens BCD digit.
0019	96	LDA	Load direct into the accumulator
001A	2B	2B	the binary result.
001B	8B	ADD	Add immediate
001C	OA	OA	10 ₁₀ to the binary result.
001D	97	STA	Store away the new
001E	2B	2B	binary result.
001F	20	BRA	Branch
0020	F1	F1	back to the instruction in location 12 ₁₆ .
0021	96	LDA	Load direct into the accumulator
0022	2B	2B	the binary result.
0023	9B	ADD	Add direct
0024	2A	2A	the units BCD digit.
0025	97	STA	Store away the new
0026	2B	2B	binary result.
0027	3E	HLT	Halt.
0028	01	01	Hundreds BCD digit.
0029	01	01	Tens BCD digit.
002A	07	07	Units BCD digit.
002B	—	—	Reserved for the binary result.

Figure 7-38
Program for converting BCD to binary.

A strategically placed conditional branch at the end of the program can cause a branch back to the beginning that will repeat the program again and again. In the next portion of this experiment, you will load the BCD-to-binary conversion program that you studied earlier. Then you will step through the program and watch as the Trainer executes each instruction.

Procedure (continued)

23. Load the program listed in Figure 7-38 into the Trainer. The BCD number 117_{10} will be converted to binary by this program.

The BEQ instruction is used for the conditional branches in this program. This means that MPU will monitor the _____ flag to determine if the condition is set.

24. Now set the program counter to 0000 and single-step through the program recording the information in the chart of Figure 9-39. Notice that, at strategic steps, you should stop and answer questions before you continue.

25. What is the hundreds BCD digit at this time? _____. The result is now 64_{16} , which is _____ in the decimal number system.

Now return to the Trainer and continue stepping through the program.

26. What is the tens BCD digit at this time? _____.

The result is now $6E_{16}$. This is the equivalent of _____ in the decimal number system.

Now return to the Trainer and step through the remainder of the program.

27. Examine address $002B_{16}$ and record the result below.

_____16

Convert this number to its decimal equivalent.

$75_{16} = \text{_____}_{10}$

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
Stop! Return to Step 25.					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
Stop! Return to step 26.					
19					
20					
21					
22					
23					
24					
25					

Figure 7-39

Discussion

Now you've verified the operation of the BCD-to-binary conversion program. The chart that you completed should match the one shown in Figure 7-40.

Since the BEQ instruction is used for the conditional branches in the program, we monitored the Z flag. In this example, the BCD number 117_{10} was converted to its binary equivalent 75_{16} . This program will convert BCD numbers as high as 255_{10} , to their binary equivalent.

The program isn't as complicated as it might appear. The hundreds and tens BCD digits are used to set a count. Each pass through a loop decrements the BCD digit, or count, and then adds the equivalent hexadecimal positional value for that BCD digit. For example, in the hundreds conversion loop, 64_{16} is added to the binary result for each hundreds BCD digit. Hence, the BCD digit sets the count. Then the count is decremented by one and the program loops back and runs through again. When the count is zero, that BCD digit has been added the correct number of times and the program branches off to another loop. This continues until the program halts.

Stepping through the program, you found that after Step 8, the Trainer had completed one loop through the hundreds BCD portion of the program. The count was 00_{16} and the binary result was 64_{16} , or the binary equivalent of 100_{10} . On the next pass through, the program branches to the tens BCD loop.

The first loop through, the tens BCD portion of the program was completed at step 18. The binary result was $6E_{16}$, which is the equivalent of 110_{10} . The tens BCD digit had been decremented to 00_{16} . Then all that remained was to add the units BCD digit (07_{10}) and the conversion process was complete.

You verified the final result by checking the binary result at location $002B_{16}$. Here you found the hex number 75_{16} . When you converted this number to its decimal equivalent, you found that 75_{16} equals 117_{10} . Also, if you converted 75_{16} to binary, you would find the number $0111\ 0101_2$, which is the (binary) equivalent of 117_{10} , so the program works.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0001	97	00	1	Store 00 in address 002B ₁₆ . This clears the binary result.
2	0003	96	00	1	Load the accumulator with the Hundreds BCD digit (01 ₁₆).
3	0005	27	Hundreds BCD→ Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
4	0007	4A	01	0	Decrement the BCD Hundreds Digit.
5	0008	97	New→ Hundreds Digit 00	1	Store the new Hundreds Digit (00).
6	000A	96	00	1	Load the accumulator with the Binary Result (00 ₁₆).
7	000C	8B	00	1	Add to the binary result 64 ₁₆ .
8	000E	97	Binary→ Result Now 64	0	Store away the new binary result.
9	0010	20	64	0	Branch back to address 0003 ₁₆ .
10	0003	96	64	0	Load the accumulator with the Hundreds BCD digit (00).
11	0005	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0012 ₁₆ .
12	0012	96	00	1	Load the accumulator with the tens BCD digit (01 ₁₆).
13	0014	27	Tens BCD→ Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
14	0016	4A	01	0	Decrement the tens BCD digit (01 ₁₆).
15	0017	97	New Tens→ Digit 00	1	Store the new tens BCD digit.
16	0019	96	00	1	Load the accumulator with the binary result (64 ₁₆).
17	001B	8B	64	0	Add 0A ₁₆ to the binary result.
18	001D	97	New Binary→ Result 6E	0	Store away the new binary result.
19	001F	20	6E	0	Branch back to address 0012 ₁₆ .
20	0012	96	6E	0	Load the accumulator with the tens BCD digit (00).
21	0014	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0021 ₁₆ .
22	0021	96	00	1	Load the accumulator with the binary result (6E ₁₆).
23	0023	9B	6E	0	Add the units BCD digit (07 ₁₆).
24	0025	97	New Binary→ Result 75	0	Store the new binary result (75 ₁₆).
25	0027	3E	75	0	Halt.

Figure 7-40

Single-stepping through the BCD-to-binary conversion program.

The most frequent mistake made by programmers when using the branch instructions is the improper computation of the relative address. An improperly coded relative address not only prevents the program from executing properly, but can even wipe out portions of the program. In the next section of this experiment, you will witness the result of an incorrect relative address and the effect it has on the program. In this example, we will use the binary-to-BCD conversion program you studied earlier.

Procedure (continued)

28. Load the program listed in Figure 7-41 into the Trainer. This program should convert the binary number $0111\ 0101_2$ (75_{16}) into its BCD equivalent. However, one of the relative addresses is **incorrect**. Part of this exercise is to locate the incorrect relative address and correct it.
29. Now set the program counter to 0000 and single-step through the program. Record the results in the chart of Figure 7-42. Notice that we're monitoring the carry (C) flag because the program uses the BCS (Branch if Carry Set) instruction.
30. Examine addresses $002B_{16}$, $002C_{16}$, and $002D_{16}$; record the results below.

002B ___ ___ Hundreds BCD Digit

002C ___ ___ Tens BCD Digit

002D ___ ___ Units BCD Digit

Obviously, there is something wrong with the program. Although the hundreds and tens digits are believable, the units digit of 11 is impossible. Remember, a decimal number can only have a units digit of from 0 to 9_{10} .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	STA	Store 00
0002	2B	2B	in location 002B ₁₆ . This clears the hundreds digit.
0003	97	STA	Store 00.
0004	2C	2C	in location 002C ₁₆ . This clears the tens digit.
0005	97	STA	Store 00
0006	2D	2D	in location 002D ₁₆ . This clears the units digit.
0007	96	LDA	Load direct into the accumulator
0008	2A	2A	the binary number to be converted.
0009	80	SUB	Subtract immediate
000A	64	64	100 ₁₆ .
000B	25	BCS	If a borrow occurred, branch
000C	0A	0A	forward to the instruction in location 0016 ₁₆ .
000D	97	STA	Otherwise, store the result of the subtraction
000E	2A	2A	as the new binary number.
000F	96	LDA	Load direct into the accumulator
0010	2B	2B	the hundreds digit of the BCD result.
0011	4C	INCA	Increment the hundreds digit.
0012	97	STA	Store the hundreds digit
0013	2B	2B	back where it came from.
0014	20	BRA	Branch
0015	F1	F1	back to the instruction at address 0007 ₁₆ .
0016	96	LDA	Load direct into the accumulator
0017	2A	2A	the binary number.
0018	80	SUB	Subtract immediate
0019	0A	0A	10 ₁₆ .
001A	25	BCS	If a borrow occurred, branch
001B	09	09	forward to the instruction in location 0025 ₁₆ .
001C	97	STA	Otherwise, store the result of the subtraction
001D	2A	2A	as the new binary number.
001E	96	LDA	Load direct into the accumulator
001F	2C	2C	the tens digit.
0020	4C	INCA	Increment the tens digit.
0021	97	STA	Store the tens digit.
0022	2C	2C	back where it came from.
0023	20	BRA	Branch
0024	F1	F1	back to the instruction at address 0016 ₁₆ .
0025	96	LDA	Load direct into the accumulator
0026	2A	2A	the binary number.
0027	97	STA	Store it in
0028	2D	2D	the units digit.
0029	3E	HLT	Halt.
002A	75	75	Place binary number to be converted at this address.
002B	—	—	Hundreds digit
002C	—	—	Tens digit
002D	—	—	Units digit

} Reserved for
BCD result.

Figure 7-41
A program with an incorrect relative address.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure 7-42

Single-Stepping through the binary-to-BCD conversion program.

31. Use the program listing and the chart that you've compiled and locate the error in the program. Then record the address of the instruction below.

HINT: The problem is with the relative address for one of the branch instructions. When one of these addresses is incorrect, the program branches to the wrong address, possibly skipping portions of the program. Therefore, first determine the portions of the program that produced the wrong result and work back until you find the problem.

Address _ _ _ _ Incorrect Relative Address _ _

32. Now calculate the correct relative address (operand) and record it below.

Correct Relative Address _ _.

Discussion

This exercise should have demonstrated the versatility of your Trainer to assist you in “debugging” programs. When you examined addresses 002B₁₆, 002C₁₆, and 002D₁₆, you found these results.

002B	<u>0</u>	<u>1</u>	Hundreds BCD Digit
002C	<u>0</u>	<u>0</u>	Tens BCD Digit
002D	<u>1</u>	<u>1</u>	Units BCD Digit

Obviously, the units BCD digit is incorrect. Since the units digit is wrong, we begin to debug at this portion of the program. This happens to be the least complex section of the program because the binary number is simply loaded into the accumulator and stored in address 002D₁₆. Comparing the chart that you compiled against the program listing, we find that this portion of the program seems to be executing correctly.

Therefore, we move back to the tens BCD digit portion of the program. Checking the program listing, we find that the tens BCD portion of the program begins at address 0016₁₆. But as the chart in Figure 7-43 shows, when the program is single-stepped the tens BCD digit loop actually starts at address 0017₁₆. This is the wrong address. We find the problem when we move back to step 14 of the chart. This is the BCS (Branch if Carry Set) instruction at address 000B₁₆. However, instead of branching to address 0016₁₆ as the comments column suggests, the program goes to address 0017₁₆. Therefore, the relative address at address 000C₁₆ must be incorrect. When we check this relative address, we find that it should be 09₁₆, instead of 0A₁₆.

But, how did this incorrect operand affect the program? Following the chart in Figure 7-43, we find that the hundreds BCD portion of the program worked correctly. On the second loop through this portion of the program, the subtraction resulted in a borrow and the C flag was set. Hence, the BCS instruction produced the desired branch.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	97	00	0	Store 00 in Hundreds Digit.
2	0003	97	00	0	Store 00 in tens Digit.
3	0005	97	00	0	Store 00 in units Digit.
4	0007	96	00	0	Load the accumulator with the Binary number (75 ₁₆).
5	0009	80	75	0	Subtract 64 ₁₆ from accumulator
6	000B	25	11	0	BCS. Check C flag for borrow. It's clear so continue.
7	000D	97	11	0	Store away the new binary number.
8	000F	96	11	0	Load the accumulator with the Hundreds Digit (00).
9	0011	4C	00	0	Increment the Hundreds Digit.
10	0012	97	01	0	Store the Hundreds Digit.
11	0014	20	01	0	Branch back to address 0007 ₁₆ .
12	0007	96	01	0	Load the accumulator with the Binary Number (11 ₁₆).
13	0009	80	11	0	Subtract 64 ₁₆ from accumulator. BCS. Check C Flag for borrow.
14	000B	25	AD	1	It's set so branch to address 0016 ₁₆ .
15	<div>Tens BCD</div> <div>0017</div>	<div>Wrong Address 2A</div> <div>AD</div>	AD	1	What's this?
16	0019	0A	AD	1	
17	001A	25	AD	1	BCS. Check C Flag. It's still set so branch to address 0025 ₁₆ .
18	<div>Units BCD</div> <div>0025</div>	96	AD	1	Load the accumulator with the Binary number.
19	0027	97	11	1	Store it in the units Digit.
20	0029	3E	11	1	Halt.

Figure 7-43
Locating the incorrect relative address.

But, instead of branching to address 0016_{16} , where we would have found a load accumulator instruction (96_{16}) with an operand of $2A_{16}$, the program branches to address 0017_{16} . The Trainer now interprets the operand ($2A_{16}$) as an instruction or op code. The op code $2A$, as you may recall, represents a valid instruction which is "Branch if Plus." The MPU checks the N flag and finds it set, because at this time, the negative number AD_{16} is in the accumulator. Therefore, the condition is not satisfied, and the Trainer continues on to the next instruction.

Single-stepping again (now we are at step 16) the next op code is $0A$. Actually, this should be the operand for the subtract instruction at address 0018_{16} . But since we are off by one, it appears to be the op code. The Trainer checks the op code $0A$ and finds that it's an inherent instruction to "clear the overflow flag." It executes this instruction.

Step 17 finds the program at address $001A_{16}$. Here, we encounter another BCS conditional branch instruction. The C flag is still set so we branch to address 0025_{16} . The program works properly from this point on.

Therefore, this one incorrect relative address caused the program to skip the tens BCD portion of the program. The tens unit was never subtracted, so it carried over into the units BCD digit. This produced the wrong units digit of 11_{10} .

Procedure (continued)

33. Now change the operand at address $000C_{16}$ from $0A_{16}$ to 09_{16} .
34. Also change the number at address $002A_{16}$ to 75_{16} . This is the number that the program will convert to its BCD equivalent.
35. Reset the program counter to 0000 and single-step through the program comparing the program listing with the results that you obtain.

36. Examine the addresses listed below and record the information stored there.

002B — — Hundreds BCD Digit

002C — — Tens BCD Digit

002D — — Units BCD Digit

Is this the correct BCD representation for the number 75_{16} ?

_____.

Discussion

When the program is corrected by inserting the relative address (09_{16}) at address $000C_{16}$, we find that it works perfectly. After single-stepping through the program, we examine the BCD digits stored at addresses $002B_{16}$, $002C_{16}$, and $002D_{16}$. The hundreds digit is 01_{10} , the tens digit is 01_{10} , and the units digit is 07_{10} . Therefore, the BCD equivalent of the binary number $0111\ 0101_2$ (75_{16}) is 117_{10} .

EXPERIMENT 6

Additional Instructions

OBJECTIVES:

To verify the operation of the ADC instruction when used in a multiple-precision addition program.

To investigate the hazard of using the ADC instruction when a carry is not desired.

To demonstrate your ability to write a multiple-precision subtraction program using the SBC instruction.

To demonstrate your ability to write a routine that will multiply any 4-bit binary number times 16_{10} using the ASLA instruction.

To verify the operation of a BCD packing program that uses the ASLA instruction.

To verify the operation of the DAA instruction when used in a BCD multiple-precision addition program.

Introduction

One of the measures of a microprocessor's power is the size of the instruction set. In other words, more instructions generally mean more potential power. You saw the economy that resulted with the addition of branch instructions in the previous experiment. In this experiment, we will examine four additional instructions; the ADC or add with carry, the SBC or subtract with carry, the ASLA or arithmetic shift accumulator left, and the DAA or decimal adjust accumulator.

The Discussion in Unit 4 explained the purpose of each instruction. In this experiment, we will restrict our activity to verifying that each instruction works as explained.

In the previous experiment, you examined the condition code registers and how the MPU monitors these flag registers to initiate conditional branches. Yet, these condition code registers are also monitored for other instructions. For example, the ADC (add with carry) and SBC (subtract with carry) instructions key on the C or carry flag. If an ADC instruction is executed and the carry flag is set, one is added to the least significant bit in the accumulator. Likewise, if the C flag is set when an SBC instruction is executed, one is subtracted from the least-significant bit of the accumulator. Remember, the C flag represents a "borrow" to the subtract instruction.

In the first portion of this experiment, we will verify the operation of the ADC instruction with a program for multiple precision arithmetic. Then we will examine one of the hazards of using this instruction.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	No operation
0001	96	LDA	Load the accumulator direct with the
0002	0E	OE	least significant byte of the addend.
0003	9B	ADD	Add direct the
0004	10	10	least significant byte of the augend.
0005	97	STA	Store the result in the
0006	12	12	least significant byte of the sum.
0007	96	LDA	Load the accumulator direct with the
0008	0F	OF	most significant byte of the addend.
0009	99	ADC	Add with carry direct the
000A	11	11	most significant byte of the augend.
000B	97	STA	Store the result in the
000C	13	13	most significant byte of the sum.
000D	3E	HLT	Halt
000E	EA	EA	Least significant byte
000F	CO	CO	Most significant byte
			} addend
0010	93	93	Least significant byte
0011	1B	1B	Most significant byte
			} augend
0012	—	—	Least significant byte
0013	—	—	Most significant byte
			} sum

Figure 7-44
Program for multiple-precision addition.

Procedure

1. Turn on the Trainer and press the RESET key.
2. Load the program listed in Figure 7-44 into the Trainer. This program performs multiple-precision addition of two 16_{10} bit numbers. The augend $1B93_{16}$ will be added to the addend $COEA_{16}$ by this program. Of course, the program can add any numbers that are 16_{10} bits or less.
3. Change the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 7-45. Notice that we are monitoring the carry (C) flag.
4. Examine memory location 0012_{16} and 0013_{16} and record the sum below.

SUM _ _ _ _

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 7-45

5. Add the binary numbers below. These numbers are the binary equivalent of the two hex numbers added by the program just executed.

		MSB		LSB
COEA ₁₆	=	1100	0000	1110 1010
1B93 ₁₆	=	0001	1011	1001 0011
SUM	=			

Now, convert the binary sum to its hexadecimal equivalent and record below.

SUM _ _ _ _

Does this match the sum obtained in step 4? _____

6. Now load the program of Figure 7-46 into the Trainer. This program simply adds two binary numbers and produces a carry. Hence, it will set the C flag. You will see its purpose in a moment.

Execute the program by pressing the DO key and then entering address 0000.

7. Examine the carry (C) condition code register. The C flag is _____
set/reset

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load the accumulator immediate
0001	EA	EA	with EA ₁₆ .
0002	8B	ADD	Add immediate
0003	93	93	93
0004	3E	HLT	Halt

Figure 7-46
Program adds two numbers and produces carry.

8. Enter the program listed in Figure 7-47 into the Trainer. Notice that this is the same multiple-precision addition program previously executed, with the exception that the ADD Instruction has been replaced by the ADC instruction, as shown by the shaded section.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	No operation
0001	96	LDA	Load the accumulator direct with the
0002	0E	0E	least significant byte of the addend
0003	99	ADC	Add with carry direct the
0004	10	10	least significant byte of the augend.
0005	97	STA	Store the result in the
0006	12	12	least significant byte of the sum.
0007	96	LDA	Load the accumulator direct with the
0008	0F	0F	most significant byte of the addend.
0009	99	ADC	Add with carry direct the
000A	11	11	most significant byte of the augend.
000B	97	STA	Store the result in the
000C	13	13	most significant byte of the sum.
000D	3E	HLT	Halt
000E	EA	EA	Least significant byte
000F	CO	CO	Most significant byte
			} addend
0010	93	93	Least significant byte
0011	1B	1B	Most significant byte
			} augend
0012	—	—	Least significant byte
0013	—	—	Most significant byte
			} sum

Figure 7-47

Multiple-precision addition program with instruction at address 0003₁₆ changed.

9. Set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 7-48.
10. Examine memory locations 0012₁₆ and 0013₁₆. Record the sum below.

SUM _ _ _ _

Compare this sum to the previous sum recorded in step 4. Are they the same? _____.
yes/no

Why are the sums different? _____

From this demonstration, what conclusion can you draw concerning the use of the ADC instruction? _____

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 7-48

Discussion

In steps 1 through 3 of this experiment, you loaded a multiple-precision addition program similar to the one you studied in Unit 4. Single-stepping through the program, you witnessed the operation of the ADC instruction. The chart you compiled should be similar to the chart in Figure 7-49. When you checked memory locations 0012_{16} and 0013_{16} , you found the LSB and MSB respectively of the 16_{10} -bit sum. The sum should have been $DC7D_{16}$.

In step 5 you added the binary equivalents of the hex numbers, $COEA_{16}$ and $1B93_{16}$. The sum was the binary equivalent of the sum produced by the program, as shown below.

		MSB	LSB
		1	
$COEA_{16}$	=	1100 0000	1110 1010
$1B93_{16}$	=	0001 1011	1001 0011
SUM	=	1101 1100	0111 1101

As you noticed, a carry is generated by the addition of the least significant bytes of the two numbers. When you were single-stepping through the program, you observed this carry because the C flag was set. The addition of the most significant bytes did not produce a carry. Therefore, the carry flag was cleared.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	Random	Load the accumulator with the LSB of Addend (EA_{16}).
2	0003	9B	EA	Random	Add the LSB of the Augend (93_{16}).
3	0005	97	7D	1	Store result in LSB of sum.
4	0007	96	7D	1	Load the accumulator with the MSB of the Addend (CO_{16}).
5	0009	99	CO	1	Add with carry the MSB of the Augend ($1B_{16}$).
6	000B	97	DC	0	Store result in MSB of Sum.
7	000D	3E	DC	0	Halt.

Figure 7-49

When you converted the binary number to hexadecimal, you found that the sum was the same as that produced by the program.

1101 1100 0111 1101

D C 7 D

In step 6, you loaded a simple program that added the numbers EA_{16} and 93_{16} . Of course, the addition generated a carry, as you witnessed when you checked the C flag and found it set.

In step 8, you loaded another multiple-precision addition program into the Trainer. The only difference between this program and the previous multiple-precision addition program was that the first add instruction was the ADC (add with carry), rather than the ADD. Then you single-stepped through the program and completed the chart of Figure 7-48. Your chart should be similar to the one shown in Figure 7-50.

When you examined the sum at addresses 0012_{16} and 0013_{16} , you found $DC7E_{16}$. The correct sum, as you verified earlier, should have been $DC7D_{16}$. If you checked the chart compiled while single-stepping through the program, the reason for this incorrect answer should have been evident. The carry flag was set even before the program was executed. Therefore, when the Trainer executed the first ADC instruction, it automatically added the carry (1_2) to the sum of the least significant bytes. Hence, the result $7E$ was one greater than the correct sum of $7D$.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	1	Load the accumulator with the LSB of Addend (EA_{16}).
2	0003	99	EA	1	Add with carry the LSB of the Augend 93_{16} .
3	0005	97	7E	1	Store result in LSB of sum. Load the accumulator with the MSB
4	0007	96	7E	1	of Addend (CO_{16}).
5	0009	99	CO	1	Add with carry the MSB of the Augend ($1B_{16}$).
6	000B	97	DC	0	Store result in MSB of sum.
7	000D	3E	DC	0	Halt.

Figure 7-50

Single-stepping through the multiple-precision addition program where both add instructions are ADC.

From this demonstration you should have reached the conclusion that the ADC instruction should not be used unless you are positive of the condition of the C flag. You must remember that the C flag is only reset by an arithmetic operation that doesn't produce a **carry** or a **borrow**. For example, in the program that worked properly, we used the simple ADD instruction for the first addition. Naturally, this instruction ignores the condition of the C flag, so it doesn't matter if it's set or reset. This is a simple way of playing it safe. The second addition used the ADC instruction because we wanted any carry from the least significant byte to be reflected in the most significant byte.

The SBC (subtract with carry) instruction is similar to the ADC instruction because it also monitors the C flag to indicate a borrow. In the next section of this experiment, you will write a program that uses the SBC instruction for multiple-precision subtraction of 16_{10} -bit numbers.

Procedure (continued)

11. Write a program that will perform multiple-precision subtraction of two 16_{10} -bit (2-byte) numbers. The following guidelines define the problem.
 - a. The program must subtract a 16_{10} -bit subtrahend from a 16_{10} -bit minuend and store the difference in memory.
 - b. Use the direct addressing mode.
 - c. Select the op codes from the instruction listing in Figure 7-51.
12. Now load the program. Enter 9721_{16} in the locations reserved for the minuend and 7581_{16} in the locations reserved for the subtrahend.
13. Single-step through the program and observe its operation. Examine the locations where the difference is stored and record the 2-byte difference below.

DIFFERENCE _____

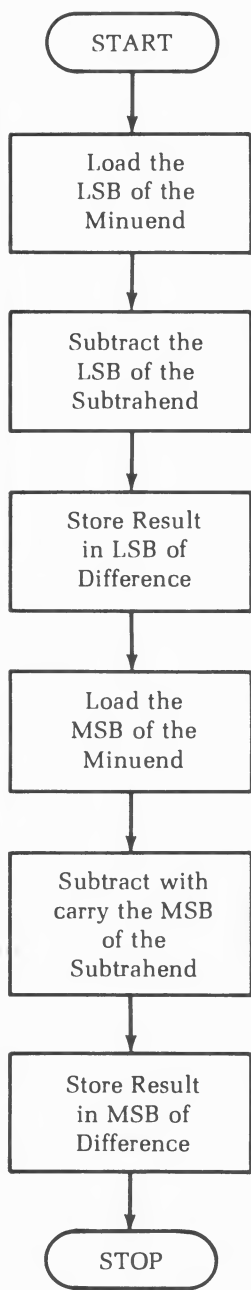


Figure 7-52
Flow chart for
multiple-precision subtraction.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		4F
Clear Accumulator	CLRA				4A
Decrement Accumulator	DECA				4C
Increment Accumulator	INCA				
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Add with Carry	ADC	89	99		
Subtract with Carry	SBC	82	92		
Arithmetic Shift					
Accumulator Left	ASLA				48
Decimal Adjust					
Accumulator	DAA				19
Halt	HLT				3E

Figure 7-51
Instructions.

Discussion

If you made a flow chart of the problem, your flow chart probably looks like the one shown in Figure 7-52. Your program should be similar to the solution shown in Figure 7-53. After stepping through the program on the Trainer, the difference of the subtraction should have been $21A0_{16}$. If you didn't obtain this answer, go back and recheck your program.

You may have used the SBC instruction for the first subtraction. If you did, this might explain the problem, because if the C flag is set when this instruction is executed a 1 will be borrowed from the difference. Therefore, your answer would have been 1 less than the correct answer, or $219F_{16}$. If the carry flag was cleared before you executed the program, the result would still be correct.

In the next section of this experiment, we will examine the ASLA (arithmetic shift accumulator left) instruction. You will also write a simple program that uses this instruction to multiply any 4_{10} -bit number by 16_{10} . This simple routine will prove it's usefulness later.

Recall from the discussion in Unit 4 that each ASLA operation multiplies the contents of the accumulator by two.

Procedure (continued)

- Use the instructions listed in Figure 7-51 and write a program that uses the ASLA instruction to multiply any 4_{10} -bit number by 16_{10} .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	0D	0D	least significant byte of minuend
0002	90	SUB	Subtract direct
0003	0F	0F	least significant byte of subtrahend
0004	97	STA	Store result in
0005	11	11	least significant byte of difference
0006	96	LDA	Load accumulator direct with
0007	0E	0E	most significant byte of minuend
0008	92	SBC	Subtract with carry
0009	10	10	most significant byte of the subtrahend
000A	97	STA	Store result in
000B	12	12	most significant byte of difference
000C	3E	HLT	Halt
000D	21	21	Least significant byte
000E	97	97	Most significant byte
000F	81	81	Least significant byte
0010	75	75	Most significant byte
0011	—	—	Least significant byte
0012	—	—	Most significant byte

Figure 7-53

Program for multiple-precision subtraction.

15. Enter your program into the Trainer and then have your program multiply OF_{16} (15_{10}) by 16_{10} . Record the product below.

$$OF_{16} \times 16_{10} = \text{————}_{16}$$

16. Convert the product obtained to its decimal equivalent.

$$\text{Decimal equivalent } \text{————}_{10}$$

Now check your result by multiplying 15_{10} times 16_{10} .

$$15_{10} \times 16_{10} = \text{————}_{10}$$

17. In this program, the multiplier is determined by the number of ASLA instructions. How many ASLA instructions are required to produce a multiplier of 4_{10} ? _____.

Discussion

The program for this simple routine is shown in Figure 7-54. Notice that it uses 4_{10} ASLA instructions to produce the required multiplier of 16_{10} . If your program worked properly, the final product should have been $F0_{16}$. Converting this number to its decimal equivalent, we find that $F0_{16}$ equals 240_{10} . When we multiplied 15_{10} times 16_{10} , we also found the product was 240_{10} . Therefore, the program works.

Only two ASLA instructions are necessary to produce a multiplier of 4_{10} ; three ASLA instructions will result in a multiplier of 8_{10} .

Another use for the ASLA instruction is to pack two BCD digits into a single byte. This "packing" can result in a significant savings of memory if many BCD numbers are used. Let's verify the operation of the BCD packing program that was presented in Unit 4.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator with the 4-bit multiplicand
0001	09	09	
0002	48	ASLA	} Shift the accumulator four places to the left multiplying the multiplicand by 16_{10} .
0003	48	ASLA	
0004	48	ASLA	
0005	48	ASLA	
0006	97	STA	Store the product at this location
0007	0A	0A	
0008	3E	HLT	Halt
0009	0F	0F	4-bit multiplicand
000A	—	—	Product

Figure 7-54

Program that uses the ASLA instruction to multiply a 4-bit number times 16_{10} .

Procedure (continued)

18. Enter the BCD packing program listed in Figure 7-55 into the Trainer. The unpacked BCD numbers are 09_{10} and 03_{10} .
19. Set the program counter to 0000 and single-step through the program, recording the information below. Where it is indicated, convert the hexadecimal contents of the accumulator to the binary equivalent.

Program Count	Op code	ACCA	Binary Equivalent
0001	96	Random	Random
0003	48	_____	_____
0004	48	_____	_____
0005	48	_____	_____
0006	48	_____	_____
0007	9B	_____	_____
0009	97	_____	_____
000B	3E	HALT	

HEX ADDRESS	OPCODES/ CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	Do nothing
0001	96	LDA	Load into the accumulator direct
0002	0D	0D	the unpacked most significant BCD digit.
0003	48	ASLA	} Shift it four places to the left.
0004	48	ASLA	
0005	48	ASLA	
0006	48	ASLA	
0007	9B	ADD	Add the
0008	0E	0E	unpacked least significant BCD digit.
0009	97	STA	Store the result
000A	0C	0C	in the packed BCD number
000B	3E	HLT	Halt
000C	00	00	Packed BCD number
000D	09	09	Unpacked most significant BCD digit.
000E	03	03	Unpacked least significant BCD digit.

Figure 7-55
Program to pack two BCD digits into a single byte.

20. Examine the packed BCD number at address 000C₁₆ and record it below.

Packed BCD Number _____

Discussion

As you can see, the BCD packing program is very simple. Nevertheless, simple routines such as this can be combined in many programs, easing the task of programming. Most programmers either commit these general purpose routines to memory or file them away for future reference.

The results you obtained by stepping through the program should be similar to those shown below.

PROGRAM COUNT	OP CODE	ACCA	BINARY EQUIVALENT
0001	96	Random	Random
0003	48	09	0000 1001
0004	48	12	0001 0010 After 1st shift
0005	48	24	0010 0100 After 2nd shift
0006	48	48	0100 1000 After 3rd shift
0007	9B	90	1001 0000 After 4th shift
0009	97	93	1001 0011
000B	3E		

As the listing shows, the most significant BCD digit (09₁₀) is loaded into the accumulator. Four ASLA shifts take place, moving this digit progressively to the left. Following these four shifts, the most significant BCD digit is properly positioned. Now the program simply adds the least significant BCD (03₁₀) to the contents of the accumulator and then stores the sum. Checking the address of the packed BCD number, we find 93₁₀.

When BCD numbers are added, we encounter yet another problem. Often, the sum is the correct BCD number. But, just as frequently, it isn't. In Unit 4, the reason for this inconsistency was discussed. However, your Trainer has an instruction, called the "Decimal Adjust Accumulator" (DAA), that can correct the sum of BCD numbers, producing the desired result.

In the next portion of this experiment, we will demonstrate the need for the DAA instruction by first adding two BCD numbers without using the DAA instruction. Then we will check the sum. Next, we will correct the program by inserting DAA instructions and again examine the BCD sum.

Procedure (continued)

21. Load the program listed in Figure 7-56 into your Trainer. This program adds the BCD numbers 3792_{10} and 5482_{10} , storing the sum in address 0011_{16} and 0012_{16} .
22. RESET the Trainer and execute the program by first pressing the DO key and entering address 0000.
23. Again, press the RESET key and then examine the sum stored at address 0011_{16} and 0012_{16} . The most significant byte of the sum is at address 0011_{16} and the least significant byte is at address 0012_{16} . Record the sum below.

SUM _____

Is this the correct BCD sum for the addition of the numbers 3792_{10} and 5482_{10} ? _____
yes/no

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator direct with
0001	0E	0E	the least significant byte of addend.
0002	9B	ADD	Add direct
0003	10	10	the least significant byte of augend
0004	97	STA	Store the result in
0005	12	12	the least significant byte of BCD sum.
0006	96	LDA	Load the accumulator direct with
0007	0D	0D	the most significant byte of addend
0008	99	ADC	Add with carry
0009	0F	0F	the most significant byte of augend
000A	97	STA	Store the result in
000B	11	11	the most significant byte of BCD sum.
000C	3E	HLT	Halt
000D	37	37	Most significant byte
000E	92	92	Least significant byte
000F	54	54	Most significant byte
0010	82	82	Least significant byte
0011	—	—	Most significant byte
0012	—	—	Least significant byte

Figure 7-56

Incorrect program for multiple-precision addition of BCD numbers.

24. Now load the corrected multiple-precision BCD addition program listed in Figure 7-57 into your Trainer. Notice that the only changes between this program and the previous program are the additions of the NOP instruction and the two DAA instructions following the addition operations.
25. Change the program counter to 0000 and single-step through the program, recording the information below.

STEP 1

PROGRAM COUNT	OP CODE
---------------	---------

STEP 2

PROGRAM COUNT	OP CODE	ACCA
---------------	---------	------

STEP 3

PROGRAM COUNT	OP CODE	ACCA	C FLAG
---------------	---------	------	--------

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	Do nothing
0001	96	LDA	Load the accumulator direct with the
0002	11	11	least significant byte of addend.
0003	9B	ADD	Add direct
0004	13	13	the least significant byte of augend.
0005	19	DAA	Decimal adjust the sum to BCD.
0006	97	STA	Store the result in the
0007	15	15	least significant byte of BCD sum
0008	96	LDA	Load the accumulator direct with the
0009	10	10	most significant byte of addend.
000A	99	ADC	Add with carry the
000B	12	12	most significant byte of augend.
000C	19	DAA	Decimal adjust the sum to BCD.
000D	97	STA	Store the result in the
000E	14	14	most significant byte of BCD sum.
000F	3E	HLT	Halt.
0010	37	37	Most significant byte
0011	92	92	Least significant byte
0012	54	54	Most significant byte
0013	82	82	Least significant byte
0014	—	—	Most significant byte
0015	—	—	Least significant byte

BCD Addend	}	Most significant byte
		Least significant byte
BCD Augend	}	Most significant byte
		Least significant byte
BCD Sum	}	Most significant byte
		Least significant byte

Figure 7-57
Program for adding multiple-precision BCD numbers.

The sum of the addition of the least significant bytes is now in the accumulator. Is this the correct BCD sum for the numbers 92_{10} and 82_{10} ? _____
yes/no

When the DAA instruction (op code 19) is executed, will this number be corrected? _____.
yes/no

STEP 4 _____
 PROGRAM COUNT OP CODE ACCA C FLAG

As you can see, the DAA instruction did correct the left-most digit by adding 60_{16} to the sum. Since the result 14_{10} appears to be a legitimate BCD number, how did the MPU know it was not the valid BCD sum? _____

STEP 5 _____
 PROGRAM COUNT OP CODE ACCA C FLAG

STEP 6 _____
 PROGRAM COUNT OP CODE ACCA C FLAG

STEP 7 _____
 PROGRAM COUNT OP CODE ACCA C FLAG

It's obvious that this number ($8C_{16}$) is not the BCD sum of 37_{10} and 54_{10} . What number will the MPU add to $8C_{16}$ to produce the desired BCD sum? _____.

STEP 8 _____
 PROGRAM COUNT OP CODE ACCA C FLAG

STEP 9 _____
 PROGRAM COUNT OP CODE ACCA

26. Now examine the BCD sum at addresses 0014₁₆ and 0015₁₆ and record below.

SUM _____₁₀.

Discussion

When you executed the first program to add BCD numbers, it was obvious that the sum 8C14 was not the correct BCD number. The answer should have been 9274₁₀. Naturally, the MPU considered these BCD numbers as hexadecimal numbers, hence, the hexadecimal sum.

However, when the program was modified by the addition of DAA (decimal adjust accumulator) instructions after each addition operation, the result was the correct BCD number. As you stepped through the program you saw the DAA instruction in operation.

At step 3, the BCD numbers 92₁₀ and 82₁₀ had been added and the accumulator was supposedly storing the sum 14₁₀. A carry was generated by the setting of the C flag. However, the sum was not correct. Instead of 14₁₀, the sum should have been 174₁₀. To the MPU, the addition looked something like this.

	1001	0010 ₂	= 92 ₁₆
C FLAG	1000	0010 ₂	= 82 ₁₆
	<hr/>		
1 Carry	0001	0100 ₂	114 ₁₆

If we ignore the carry, the sum 14₁₆ appears to be a legitimate BCD number. Nevertheless, the sum would be incorrect. Taking the carry flag into consideration, remember it's just an extension of the accumulator, we find the sum is 114₁₆. In hex, this is the correct sum of the two numbers.

In step 4, the DAA instruction had been executed and, as you witnessed, the number 14_{16} had been adjusted to the correct BCD sum of 74_{10} . The carry flag was set, indicating that the sum of the two left-most 4-bit binary numbers was larger than 1001_2 (9_{16}). Actually, it was $1\ 0001_2$. When the DAA instruction was executed, the MPU followed the conversion rules and adjusted the sum by adding 60_{16} as shown below.

Carry					Carry	
1	0001	0100 ₂	=		1	14_{16}
	0110	0000 ₂	=			60_{16}
1	0111	0100 ₂	=		1	74_{16}

The result is 74_{16} with a carry of 1_{16} . This is the correct BCD sum for the two BCD numbers. If we include the carry, the result is 174_{10} which is indeed the decimal sum of 92_{10} and 82_{10} . However, this exceeds the capacity of our storage locations, since they're only 8-bits long, so the carry is carried forward to the addition of the most significant bytes of the numbers in the next step.

As you continued single-stepping through the program, the most significant bytes were loaded and added with the ADC instruction. At step 7, the sum of this addition was in the accumulator. It was obvious that the sum $8C_{16}$ wasn't a BCD number. To adjust this number to the correct BCD sum, 06_{16} was added by the DAA instruction. The BCD adjusted sum 92_{10} was the result.

In the final step of the experiment, you verified program operation by examining the BCD sum at locations 0014_{16} and 0015_{16} . Here you should have found the sum 9274_{10} .

EXPERIMENT 7

New Addressing Modes

OBJECTIVES:

To gain experience using the instruction set and registers of the 6808 MPU.

To demonstrate the indexed addressing mode.

Introduction

In Unit 5 you learned about the two new addressing modes called indexed and extended. You can use either of these modes to reach operands anywhere in memory. The indexed mode, as you saw, is especially powerful when numerous operands in consecutive memory locations are involved in the execution of a program.

This experiment will demonstrate the use of the indexed addressing mode to you; and help you acquire additional experience programming your trainer.

Procedure

1. Figure 7-58 shows a program for adding a list of ten numbers using the indexed addressing mode. Load this program into your trainer and verify that you have loaded it correctly.

HEX ADDRESSES	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0030	4F	CLRA	Clear accumulator A
0031	CE	LDX#	Load the index register immediately
0032	00	00	with the address of
0033	20	20	the first number in the list.
0034	AB	→ ADDA, X	Add to accumulator A indexed
0035	00	00	with 00 offset.
0036	08	INX	Increment index register.
0037	8C	CPX#	Compare the index register immediately
0038	00	00	with one greater than the address
0039	2A	2A	of the last number in the list.
003A	26	BNE	If there is no match
003B	F8	← F8	branch back to here.
003C	3E	WAI	Otherwise, halt.
0020	01	01	First number.
0021	02	02	Second number.
0022	03	03	Third number.
0023	04	04	
0024	05	05	—
0025	06	06	—
0026	07	07	—
0027	08	08	
0028	09	09	
0029	0A	0A	Tenth number.

Figure 7-58

Using indexed addressing to add a list of numbers.

Procedure (continued)

2. Execute the program using the single-step mode. After each step, record the contents of the program counter, accumulator A, and the index register in Figure 7-59.
3. Refer to the instruction set summary card. How many machine cycles are required to execute the program shown in Figure 7-58
_____ .

Discussion

This example illustrates that when a repetitive task is to be done, indexed addressing can save many bytes of memory. In many cases, indexed addressing requires more MPU cycles and therefore, a longer time to execute. Generally, time is of little importance compared to saving a substantial number of memory bytes.

Let's look at some other ways that indexed addressing is used.

STEP NUMBER	CONTENTS AFTER EACH STEP		
	PC	ACCA	INDEX
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			

Figure 7-59
Record values here.

Procedure (continued)

4. Write a program that will clear memory locations 0020₁₆ through 00A0₁₆. It should use indexed addressing.
5. When you are sure your program is correct, load it into the Trainer. Verify that you loaded it correctly; then execute it using the DO command.
6. Examine memory locations 0020₁₆ through 00A0₁₆. Each should be cleared. Examine locations below 0020₁₆ and above 00A0₁₆. These locations should not be cleared.
7. Debug your program if necessary and repeat steps 5 and 6 until the desired results are obtained.

Discussion

Our solution to the problem is shown in Figure 7-60. Your solution may be similar or quite different. If it achieves the proper result and requires about the same number of bytes, then it is perfectly acceptable.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	CE	LDX#	Load index register immediately with
0001	00	00	the address of the
0002	20	20	first location to be cleared.
0003	6F	CLR, X	Clear the location whose
0004	00	00	address is indicated by the index register.
0005	08	INX	Increment the index register.
0006	8C	CPX#	Compare the number in the index
0007	00	00	register with one greater than
0008	A1	A1	the address of the last location to be cleared.
0009	26	BNE	If there is no match
000A	F8	F8	branch back to here.
000B	3E	WAI	Otherwise, stop.

Figure 7-60
Program for clearing addresses 0020₁₆ through 00A0₁₆.

We still have not demonstrated the full power of indexed addressing because we have not yet used the offset capability. Let's look at how the offset capability can be used. Figure 7-61 shows three tables. The first two tables contain signed numbers, the third is initially cleared. The entries in the first two tables are to be added and the resulting sums are to be placed in the third table. That is, the first entry in table 1 is to be added to the first entry in table 2. The resulting sum is to be stored as the first entry in table 3. the second entry in table 1 is to be added to the second entry in table 2, forming the second entry in table 3; etc.

Procedure (continued)

8. Enter the data shown in Figure 7-61 into the indicated addresses.
9. Write a program that will solve the problem described above.
10. Enter the program into the Trainer, starting at location 0000, and execute it.
11. Examine addresses 0050₁₆ through 005F₁₆ to verify that the program performed properly.
12. If necessary, debug your program and try again.

TABLE 1		TABLE 2		TABLE 3	
ADDRESS	CONTENTS	ADDRESS	CONTENTS	ADDRESS	CONTENTS
0030	06	0040	FA	0050	00
0031	0F	0041	01	0051	00
0032	06	0042	1A	0052	00
0033	20	0043	10	0053	00
0034	2F	0044	11	0054	00
0035	00	0045	50	0055	00
0036	2F	0046	31	0056	00
0037	61	0047	0F	0057	00
0038	3E	0048	42	0058	00
0039	4F	0049	41	0059	00
003A	91	004A	0F	005A	00
003B	9F	004B	11	005B	00
003C	C0	004C	00	005C	00
003D	84	004D	4C	005D	00
003E	70	004E	70	005E	00
003F	E1	004F	0F	005F	00

Figure 7-61
Three tables.

Discussion

The solution to the problem is shown in Figure 7-62.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	CE	LDX#	Load index register with address
0001	00	00	of first entry
0002	30	30	in Table 1.
0003	A6	LDAA, X	Load entry from Table 1 into
0004	00	00	accumulator A.
0005	AB	ADDA, X	Add the corresponding entry from
0006	10	10	Table 2.
0007	A7	STAA,X	Store the result in the
0008	20	20	corresponding location in Table 3
0009	08	INX	Increment the index register.
000A	8C	CPX#	Compare the number in the index
000B	00	00	register with one greater
000C	40	40	than the address of the last entry in
			Table 1.
000D	26	BNE	If there is no match,
000E	F4	F4	branch to here.
000F	3E	WAI	Otherwise, stop.

Figure 7-62
Program for adding two tables.

EXPERIMENT 8

Arithmetic Operations

OBJECTIVES: *To gain practice using the instruction set and registers of the 6808 MPU.*

To demonstrate a fast method of performing multiplication.

To demonstrate a multiple-precision arithmetic.

To demonstrate an algorithm for finding the square root of a number.

To gain experience writing programs.

Introduction

In Unit 5, you were exposed to the full architecture and instruction set of the 6808 microprocessor. In this experiment, you will use some of the new-found capabilities of the microprocessors to solve simple problems.

Mathematical operations make excellent programming examples and at the same time illustrate useful procedures. For these reasons, the programs developed in this experiment are concerned with arithmetic operations.

In an earlier unit, you learned that a computer can multiply by repeated addition. However, this is a very slow method of multiplication when large numbers are used.

A much faster method of multiplying involves a shifting-and-adding process. To illustrate the procedure, consider the long hand method of multiplying two 4-bit binary numbers. The procedure looks like this.

1101 ₂	←	Multiplicand	→	13 ₁₀
1011 ₂	←	Multiplier	→	11 ₁₀
<u>1101</u>				<u>13</u>
1101				13
0000				<u>143₁₀</u>
<u>1101</u>				
10001111 ₂	←	Product	↘	

The decimal equivalents are shown for comparison purposes. The product is formed by shifting and adding the multiplicand. Put in computer terms, the procedure goes like this:

1. Clear the product.
2. Examine the multiplier. If it is 0, stop. Otherwise, go to 3.
3. Examine the LSB of the multiplier. If it is 1, add the multiplicand to the product then go to 4. If it is a 0, go to 4 without adding.
4. Shift the multiplicand to the left.
5. Shift the multiplier to the right so that the next bit becomes the LSB.
6. Go to 2.

Procedure

1. Write a program of any length that will perform multiplication in the manner indicated. Here are some guidelines:
 - A. You may use any of the instructions discussed up to this point.
 - B. To keep the program simple, only unsigned 4-bit binary numbers are to be used for the multiplier and the multiplicand.
 - C. The final product should be in Accumulator A when the multiplication is finished.
 - D. The multiplier may be destroyed during the multiplication process.
 - E. Assume that the multiplier and multiplicand are initially in memory. That is, you should load them into memory along with the program.

2. Try to write the program before you read further. If after 30 minutes, you feel you are not making progress, go on to step 3.
3. If you feel you need help, read over the following hints and then write the program.
 - A. The product should be formed in accumulator A.
 - B. The first step is to clear the product.
 - C. The multiplicand is shifted and added to Accumulator A. Accumulator B is a good place to hold the multiplicand during this process.
 - D. The multiplier can be tested for zero while still in memory by using the TST instruction followed by the BEQ instruction.
 - E. A good way to test the LSB of the multiplier is to shift the multiplier one bit to the right into the carry flag and then test the carry flag with a BCC instruction.
4. Once your program is written, load it into the Trainer and run it. Verify that it works for several different values of multipliers and multiplicands. Debug your program as necessary.

Discussion

The real test of your program is “Does it work?” If it works, then you have successfully completed this part of the experiment. One solution to the problem is shown in Figure 7-63. Compare your program with this one. If you could not write a successful program, study this program carefully to see how it handles each phase of the operation.

Obviously, this simple program has some serious drawbacks. The chief one is that the product cannot exceed eight bits. Fortunately, the basic procedure can be expanded so that much larger numbers can be handled. The solution is to use two bytes for the product. This will allow products up to $65,535_{10}$. In this example, the multiplier will be restricted to eight bits. However, the multiplicand can have up to 16 bits (two bytes) as long as the product does not exceed $65,535_{10}$. In an earlier unit, you learned that multiple-precision numbers can be added by a 2-step operation. The least significant (LS) byte of one number is added to the LS byte of the other. Then, the MS byte is added **with carry** to the MS byte of the other. Keep this in mind as you write your program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0010	4F	CLRA	Set the product to 0.
0011	D6	LDAB	Load accumulator B with the
0012	22	22	multiplicand.
0013	7D	TST	Test
0014	00	00	the
0015	23	23	multiplier.
0016	27	BEQ	If it is 0, branch to the
0017	09	09	wait instruction.
0018	74	LSR	Shift the LSB of the
0019	00	00	multiplier to the
001A	23	23	right into the carry flag.
001B	24	BCC	If the carry flag is cleared
001C	01	01	skip the next instruction.
001D	1B	ABA	Add the multiplicand to the
			product.
001E	58	ASLB	Shift the multiplicand to the left.
001F	20	BRA	Branch back and go through again.
0020	F2	F2	
0021	3E	WAI	Wait.
0022	05	Multiplicand	
0023	03	Multiplier	

Figure 7-63
Multiplying by shifting and adding.

The procedure for shifting a multiple-precision value will also come in handy. To shift a 2-byte number to the left, a 2-step procedure like that shown in Figure 7-64 can be used. First, the LS byte is shifted one place to the left into the carry bit by using the ASL instruction. Next the MS byte is rotated to the left. The result is that the 16-bit number has been shifted one bit to the left.

Procedure (continued)

5. Write a program that will multiply a double-precision multiplicand times an 8-bit multiplier. Assume that the double-precision product is to be stored in memory locations 0000_{16} and 0001_{16} . The double-precision multiplicand is initially in addresses 0002_{16} and 0003_{16} . The 8-bit multiplier is in address 0004_{16} .
6. Once again, you should try to write this program. If after 30 minutes or so you are not making progress, read the hints given in step 7.

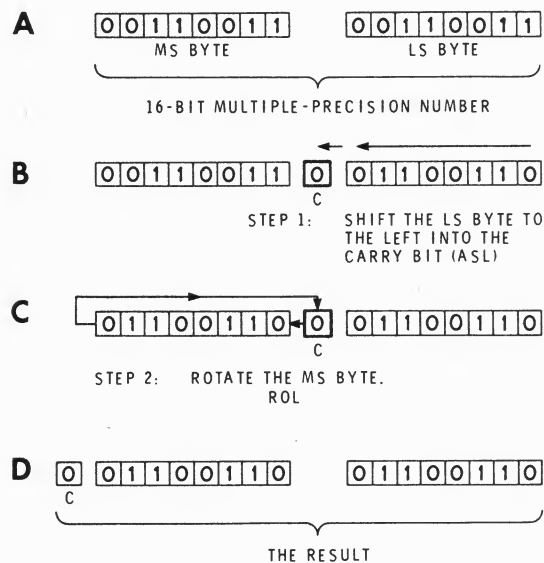


Figure 7-64
Shifting a multiple-precision number.

7. Read over the following hints (if necessary) and try again.
 - A. Initially clear both bytes of the product.
 - B. Test the multiplier for zero exactly as you did in the previous program.
 - C. Test the LSB of the multiplier as you did in the previous program.
 - D. When adding the multiplicand to the product, use the multiple-precision add technique.
 - E. When shifting the multiplicand to the left, use the technique shown in Figure 7-64.
8. Once your program is written, load it into the Trainer and verify that it works properly. Debug the program as necessary.

Discussion

There are dozens of ways in which this program could be written. If your program produces proper results, then you have been successful. One solution to the problem is shown in Figure 7-65. Compare your program with this one. If you were unsuccessful in writing a program, study Figure 7-65 very carefully until you understand the procedures involved.

Another problem that makes a good programming exercise is finding the square root of a number. Writing the program is not too difficult once you develop the proper algorithm. While there are many different ways to find the square root of a number, the easiest method from the programmer's point of view involves the subtraction of successive odd integers.

This method works because of the relationship between perfect squares. The first several perfect squares are $0^2 = 0$, $1^2 = 1$, $2^2 = 4$, $3^2 = 9$, $4^2 = 16$, $5^2 = 25$, etc. Notice:

The relationship between the numbers 0, 1, 4, 9, 16, 25, etc.

The difference between 0 and 1 is 1, the first odd integer.

The difference between 1 and 4 is 3, the second odd integer.

The difference between 4 and 9 is 5, the third odd integer; etc.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	—	—	Product (LS byte)
0001	—	—	Product (MS byte)
0002	—	—	Multiplicand (LS byte)
0003	—	—	Multiplicand (MS byte)
0004	—	—	Multiplier
*	*	*	Instructions start at address 0010
0010	7F	CLR	Clear the product.
0011	00	00	
0012	00	00	
0013	7F	CLR	
0014	00	00	
0015	01	01	
0016	7D	TST	Test the multiplier.
0017	00	00	
0018	04	04	
0019	27	BEQ	If the multiplier is 0, branch to the WAI instruction.
001A	19	19	
001B	74	LSR	Otherwise, shift the right most bit of the multiplier into the C flag.
001C	00	00	
001D	04	04	
001E	24	BCC	If the C flag is 0 branch to here.
001F	0C	0C	
0020	96	LDAA	Otherwise, load the LS byte of the product into accumulator A.
0021	00	00	
0022	9B	ADDA	Then add the LS byte of the multiplicand.
0023	02	02	
0024	D6	LDAB	Load the MS byte of the product into accumulator B.
0025	01	01	
0026	D9	ADCB	Add (with carry) the MS byte of the multiplicand.
0027	03	03	
0028	97	STAA	Store the contents of accumulator A as the LS byte of the product.
0029	00	00	
002A	D7	STAB	Store the contents of accumulator B as the MS byte of the product.
002B	01	01	
002C	78	ASL	Shift the LS byte of the multiplicand to the left.
002D	00	00	
002E	02	02	
002F	79	ROL	Rotate the MS byte of the multiplicand to the left.
0030	00	00	
0031	03	03	
0032	20	BRA	Repeat the process.
0033	E2	E2	
0034	3E	WAI	Stop.

Figure 7-65

Program for multiplying a double-precision multiplicand by an 8-bit multiplier.

This relationship gives us a simple method of finding the exact square root of perfect squares and of approximating the square root of non-perfect squares.

The procedure for finding the square root of a number looks like this:

1. Subtract successive odd integers (1, 3, 5, 7, 9, etc.) from the number until the number is reduced to 0 or a negative value.
2. Count the number of subtractions required. The count is the exact square root of the number if the number was a perfect square. The count is the approximate square root if the number was not a perfect square.

For example, let's find the square root of 49_{10} .

49	Original Number.
<u>-1</u>	Subtract the first odd integer.
48	
<u>-3</u>	Subtract the second odd integer.
45	
<u>-5</u>	Subtract the third odd integer.
40	
<u>-7</u>	Subtract the fourth odd integer.
33	
<u>-9</u>	Subtract the fifth odd integer.
24	
<u>-11</u>	Subtract the sixth odd integer.
13	
<u>-13</u>	Subtract the seventh odd integer.
0	Stop subtracting because the original number has been reduced to 0.

We simply count the number of subtractions required.

Since 7 subtractions were required, the square root of 49 is 7.

Procedure (continued)

9. With pencil and paper, use the above algorithm to find the square root of 81_{10} . Does the answer give the exact square? _____. Was the result of the final subtraction 0? _____.
10. With pencil and paper, use the above algorithm to find the square root of 119_{10} . How many subtractions are required to reduce the number to a negative value. Does this count approximate the square root of 119_{10} ? _____.
11. Write a program that uses the above algorithm to find or approximate the square root of any unsigned 8-bit number.
12. Load your program into the Trainer and run it. Verify that it works for several different values.

Discussion

Our solution to the problem is shown in Figure 7-66. The number is loaded into accumulator A, where it will be gradually reduced to a negative value. The odd integer is maintained in accumulator B. Each new odd integer is formed by incrementing twice. The SBA instruction is used to subtract the odd integer from the number. The BCS instruction is

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDAA	Load the number that is at
0001	0F	0F	this address into accumulator A.
0002	C6	LDAB#	Load accumulator B with the
0003	01	01	first odd integer.
0004	10	SBA	Subtract the odd integer from the
			number.
0005	25	BCS	If the carry is set, branch
0006	04	04	to here.
0007	5C	INCB	Otherwise, form the next higher
			odd
0008	5C	INCB	integer by incrementing B twice.
0009	20	BRA	Branch back
000A	F9	F9	to here.
000B	54	LSRB	Shift the odd integer to the right.
000C	D7	STAB	Store the answer at
000D	10	10	this address.
000E	3E	WAI	Wait.
000F	—	Number	Number to be operated upon.
0010	—	Answer	Final answer appears here.

Figure 7-66
Square root subroutine

used to determine when the number goes negative (a borrow occurs at that point). You could have used the BMI instruction but this would limit the original number to a value below $+128_{10}$. A few bytes are saved by not maintaining a separate count of the number of subtractions. Instead, the final odd integer value is converted to the count. This is possible because of the relationship between the odd integer value and the number of subtractions. As the program is written, the final odd integer is always one more than twice the number of subtractions. By shifting the final odd integer to the right, the correct count is created.

Of course, any square root program that is limited to numbers below 256_{10} is of limited use. However, this same technique can be applied to multiple-precision numbers. Figure 7-67 shows a program that can find or approximate the square root of numbers up to $16,385_{10}$. Before you study this program, try to write your own program to do this.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	I.DAA	Load accumulator A with the
0001	1A	1A	LS byte of the number.
0002	D6	I.DAB	Load accumulator B with the
0003	19	19	MS byte of the number.
0004	7F	CLR	Clear
0005	00	00	the odd
0006	1B	1B	integer.
0007	7C	→ INC	Increment.
0008	00	00	the odd
0009	1B	1B	integer.
000A	90	SUBA	Subtract the odd
000B	1B	1B	integer from the LS byte of the
			number.
000C	C2	SBCB#	Take care of any borrow
000D	00	00	from the MS byte of the number.
000E	25	BCS	If the carry is set, branch
000F	05	05	to here.
0010	7C	→ INC	Otherwise, form the next
0011	00	00	higher odd integer by
0012	1B	1B	incrementing
0013	20	BRA	and branching
0014	F2	→ F2	to here.
0015	74	→ LSR	Convert the odd integer to
0016	00	00	the answer by shifting
0017	1B	1B	right.
0018	3E	WAI	Stop.
0019	—	Number (MS)	Number to be
001A	—	Number (LS)	operated upon.
001B	—	Odd integer	Form the odd integer and the
			answer here.

Figure 7-67

Routine for finding the square root of a double precision number.

EXPERIMENT 9

Stack Operations

OBJECTIVES:

To demonstrate the stack operations that occur automatically.

To demonstrate ways that the programmer can use the stack.

To demonstrate the break-point capability of the Trainer.

Introduction

As you learned in Unit 6, the stack is used by the MPU to perform some automatic functions. When an interrupt occurs or a WAI is encountered, the MPU pushes the contents of the program counter, index register, accumulators, and condition codes on to the stack. We can easily verify this.

Procedure

- Figure 7-68 shows a program for setting the MPU registers to a known state. Examine the program and determine the hex contents of the following registers immediately after the WAI is executed.

Condition Code Register _____
 Accumulator B _____
 Accumulator A _____
 Index Register _____
 Program Counter _____

- Load the program into the Trainer and verify that you loaded it properly.
- Execute the program using the DO command.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	8E	LDS#	Load 0020 into the stack pointer
0001	00	00	
0002	20	20	
0003	CE	LDX#	Load EEDD into the index register.
0004	EE	EE	
0005	DD	DD	
0006	C6	LDAB#	Load BB into ACCB.
0007	BB	BB	
0008	86	LDAA#	Load AA into ACCA.
0009	AA	AA	
000A	36	PSHA	Push AA onto the stack.
000B	86	LDAA#	Load CC into ACCA.
000C	CC	CC	
000D	06	TAP	Transfer CC into the condition codes.
000E	32	PULA	Pull AA from the stack.
000F	3E	WAI	Wait.
0010			

Figure 7-68

This routine sets the contents of all MPU registers to known values.

4. Examine the following memory locations and record their hex contents.

Address	Contents	Register
001A	_____	_____
001B	_____	_____
001C	_____	_____
001D	_____	_____
001E	_____	_____
001F	_____	_____
0020	_____	_____

5. Identify the register from which these numbers came.
6. Try to examine the contents of ACCA, ACCB, PC, SP, and INDEX register. Do their contents agree with the number loaded there?

Discussion

When the WAI instruction is executed, the contents of the MPU registers are pushed onto the stack. Since the stack pointer is initially at 0020, the contents of the registers are stored as follows.

Address	Contents	Where it came from
001A	CC	Condition Codes
001B	BB	Accumulator B
001C	AA	Accumulator A
001D	EE	Index Register (high byte)
001E	DD	Index Register (low byte)
001F	00	Program Counter (high byte)
0020	10	Program Counter (low byte)

When you tried to examine the contents of ACCA, ACCB, SP, etc., you found that their contents did not agree with what was loaded. The reason for this **apparent** error is that the Trainer does not actually examine the contents of these registers. Instead, it examines what is placed in the stack by the WAI instruction. However, when the Trainer is reset, the monitor program assumes that the stack starts at address 00D1. Since our program moved the location of the stack, we can not use the ACCA, ACCB, PC, SP, CC, or INDEX commands after changing the stack pointer and then resetting the Trainer.

This demonstrates how the MPU uses the stack. A similar operation occurs for the SWI instruction or when a hardware interrupt occurs. Of course, the programmer can also use the stack.

Procedure (continued)

7. Figure 7-69 shows a program that will clear memory locations 0001 through 001F. It then transfers a list of numbers to these addresses. The numbers come from addresses 0051 through 006F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ ADDRESS	COMMENTS
0020	CE	LDX#	Load the index register with highest address to be cleared. Clear it.
0021	00	00	
0022	1F	1F	
0023	6F	CLR, X	Decrement index register to next lower address. Finished? If not, go back and clear the indicated address. Set index register to first entry in new list.
0024	00	00	
0025	09	DEX	
0026	26	BNE	Set the stack pointer to one less than the first entry in the old list.
0027	FB	FB	
0028	08	INX	
0029	8E	LDS#	Pull the entry from the old list. Store it in the new list.
002A	00	00	
002B	50	50	
002C	32	PULA	Increment index register to next entry in list. Finished?
002D	A7	STAA, X	
002E	00	00	
002F	08	INX	If not, go back and pull next entry. Otherwise, wait.
0030	8C	CPX#	
0031	00	00	
0032	20	20	
0033	26	BNE	
0034	F7	F7	
0035	3E	WAI	

Figure 7-69
Program for demonstrating stack operations and breakpoints.

8. Load this program into the Trainer and verify that you loaded it properly.
9. At address 0051 through 006F, load the numbers 01 through $1F_{16}$, respectively.
10. Execute the program using the DO command.
11. Examine addresses 0001 through 001F. They should contain the numbers 01 through 1F, respectively.

Discussion

This illustrates how the stack can be used in conjunction with indexing to move a list of numbers.

When this program is executed using the DO command, everything happens so fast that it is impossible to see intermediate results. Of course, you could use the single-step mode and examine the result produced by every single instruction. But in many programs, this is a long, tedious process. Therefore, the Trainer provides another way to examine programs. It allows us to set four different breakpoints in our program. The Trainer will execute instructions at its normal speed until it reaches one of these breakpoints. At that point, the Trainer will stop with the address and op code of the next instruction displayed. While the Trainer is stopped, you can examine and change the contents of any register or memory location. When you are ready to resume, you depress the return (RTI) key and the Trainer executes instructions at its normal speed until the next breakpoint or a WAI instruction is encountered.

Procedure (continued)

12. Verify that the program is still in memory.
13. Depress the RESET key. Do not depress RESET again as you perform the following steps. To do so, will erase any breakpoints that you set.
14. Refer to the program listing in Figure 7-69. Let's assume we wish to stop and examine memory and the MPU registers just before the BNE instruction at address 0026 is executed.
15. Depress the BR key. The display should be _ _ _ _ Br. The Trainer is now ready to accept the first breakpoint address. Enter the address at which the Trainer is to stop: 0026. The breakpoint is now entered.
16. Without hitting RESET, depress the DO key. Enter the address of the first instruction in the program: 0020.
17. Immediately, the display will show the address 0026 and op code 26 at which the breakpoint occurred.
18. Without hitting RESET, examine the contents of the index register. It should now read 001E.
19. Depress the EXAM key and examine address 001F. It should now be cleared.
20. Notice that you can examine the contents of any MPU register or memory location from this breakpoint mode.
21. When you are ready for the program to resume, depress the RTI key once. Again, the display will read 002626 because the MPU is back at the same breakpoint on the second pass through the first loop.
22. Examine the index register again. It should now read 001D. Examine location 001E and verify that it has been cleared.

23. The loop will be repeated 31_{10} times. On the 32^{nd} pass, the program will escape the loop.
24. Before you go further, set a second breakpoint at the INX instruction. Do this by depressing the BR key and entering the address of the instruction (0028).
25. Depress the RTI key again. Notice that the program is still stopping at the first breakpoint. It will continue to do so until it escapes the first loop.
26. You have now pushed the RTI key three times. Repeatedly push the RTI key until the display changes to 0028 08. The RTI key should have been depressed a total of 32_{10} times, counting the first three times.
27. The program is now waiting at the second break point.
28. To demonstrate a point, let's set two additional break points.
29. Depress the BR key and enter address 0029. This sets the third break point at the LDS# instruction.
30. Depress the BR key again and enter address 0033. This sets the fourth break point at the last BNE instruction.
31. The Trainer will accept only four breakpoints. We have now reached this limit. Depress the BR key again in an attempt to enter a fifth breakpoint. Notice that the word "FULL!" appears on the display.
32. Depress the RTI key so that the Trainer resumes program execution. It should stop at the third breakpoint.
33. Depress the RTI key again. The program should stop at the fourth breakpoint. Notice that the program is again in a loop. On each pass through the loop, the program will stop at this fourth breakpoint.
34. Analyze the operation of the program by examining the pertinent registers and memory locations on each pass through the loop.

Discussion

The breakpoint capability of the Trainer can be a powerful aid in writing, analyzing and debugging a program. It allows us to stop at four distinct points in the program. Here are some tips to remember when using this capability:

1. A maximum of four breakpoints can be used.
2. These may be entered all at once or during a previous breakpoint pause.
3. The RESET key erases all breakpoints.
4. The contents of the address at which the breakpoint is set must be an op code.

EXPERIMENT 10

Subroutines

OBJECTIVES: *To demonstrate the use of subroutines.*

To demonstrate that the monitor program of the ET-6800 Trainer contains some useful subroutines that can be called when needed.

To gain experience writing programs.

Introduction

Most of the subroutines that you will develop and use in this experiment deal with lighting the displays on the Trainer. For this reason, we will begin by discussing how the displays are accessed.

The ET-6800 Microprocessor Trainer has six hexadecimal displays. Each display contains eight light-emitting diodes (LEDs) arranged as shown in Figure 7-70. Each LED is given two addresses. The addresses for the left-most display are shown. To light a particular LED we simply store an odd number at the proper address. An odd number is used because the LED responds to a 1 in bit 0 of the byte that is stored. To turn an LED off, we store an even number at the proper address. The following procedure will demonstrate this.

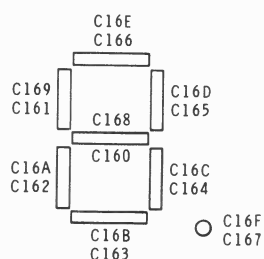


Figure 7-70

Addresses of the various segments in the left LED display.

Procedure

1. Write a program that will halt after storing an odd number (such as 01) at address C167₁₆.
2. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the decimal point of the left-most display lit.
3. Notice that the LED remains lit until it is deliberately turned off.

Discussion

To form characters, the LED's in the display must be turned on in combination. For example, to form the letter "A", the segments at addresses C162, C161, C166, C165, C164, and C160 must be turned on.

Procedure (continued)

4. Write a program that will halt after storing an odd number (such as 01) at the six addresses listed above.
5. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the letter A in the left-most display.

Discussion

Your program probably took this form:

```
LDAA  #      01
STAA  C162
STAA  C161
STAA  C166
      .
      .
      .
WAI
```

While this approach works, the program would have to be rewritten for each new character. What is needed is a program that will form many characters. One approach is to store characters as 8-bit character bytes. Since there are eight LED's in each display, each bit of the character byte can be assigned to a different LED segment. Figure 7-71A shows how

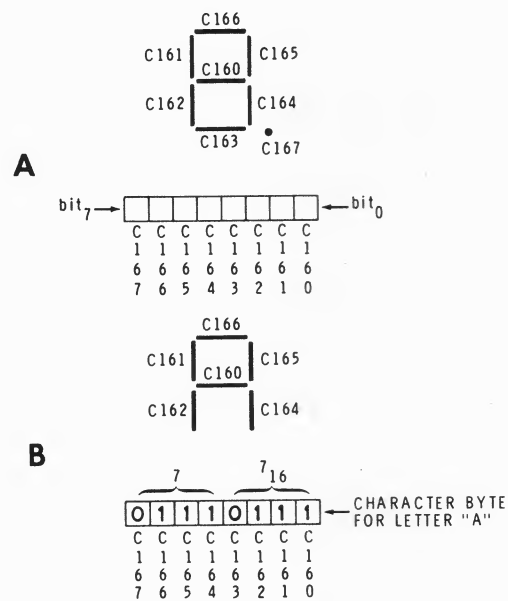


Figure 7-71

Assigning the bits of the character byte.

each bit in a character byte is assigned to each segment of the display. To light a corresponding LED, the proper bit in the character byte must be 1. For example, Figure 7-71B shows the character byte for the letter A. To form this letter, all display segments except C163 and C167 must be lit. Therefore, a 1 is placed in the character byte at all bits except the two that correspond to these addresses.

The display responds only to bit 0 of the character byte. To make each segment bit appear in turn at bit 0, the character byte must be shifted to the right. After each shift, the contents of the character byte must be stored at the address whose corresponding bit is now at bit 0. The procedure is:

1. Store the contents of the character byte at C160₁₆.
2. Shift the character byte to the right.
3. Store it at C161₁₆.
4. Shift it to the right again.
5. Store it at C162₁₆.

Etc.

A program that will do this is shown in Figure 7-72.

Procedure (continued)

6. Load the program into the Trainer and verify that you loaded it correctly.
7. Execute the program using the DO command. The left-most digit should display the letter A.
8. The character byte is at address 0001. Change this byte to 47₁₆.
9. Execute the program again using the DO command. What letter appears in the display? _____.
10. Change the character byte so that the letter H is displayed. What character byte is required? _____.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0000	86	LDAA#	Load accumulator A immediate with the
0001	77	77	character byte.
0002	CE	LDX#	Load the index register immediate with
0003	C1	C1	the address.
0004	60	60	of the left display.
0005	A7	STAA, X	Store the character byte at the
0006	00	00	address indicated by the index register.
0007	44	LSRA	Shift the character bit to the right.
0008	08	INX	Advance index register to the address of the next segment.
0009	8C	CPX	Compare index register with one greater
000A	C1	C1	than the address of the
000B	68	68	last segment.
000C	26	BNE	If no match occurs branch
000D	F7	F7	back to here.
000E	3E	WAI	Otherwise, stop.

Figure 7-72
Program for lighting a display.

11. Change the character byte to 79_{16} . Execute the program. What character is displayed? _____.
12. Refer to Figure 7-73. This figure shows the addresses of the LED's in each of the six displays. You have seen that the left display has an address of $C16X_{16}$. The X stands for some number between 0 and F, depending on which segment of that display we wish to use. The next display to the right has an address of $C15X_{16}$; etc.
13. Now return to the program in Figure 7-72. Addresses 0003 and 0004 contain the address of the affected display. By changing this address, we can move the character to a different display. Actually since all display addresses start with C1, we need only change the number at address 0004.
14. Change the byte at 0004 to 50_{16} . Change the byte at $000B_{16}$ to 58. Execute the program using the DO command. The character should appear in the second display from the left.
15. Change the byte at 0004 to 10_{16} and the byte at $000B_{16}$ to 18_{16} . Execute the program using the DO command. The character should appear in the right-most display.

Discussion

It has probably occurred to you that the monitor program must have a subroutine that performs this same function. Fortunately, this subroutine is written in such a way that we can use it. It is called OUTCH for OUTput CHaracter. It starts at address $FE3A_{16}$. We can call this subroutine anytime we like by using the JSR instruction. This subroutine assumes that the character byte is in accumulator A.

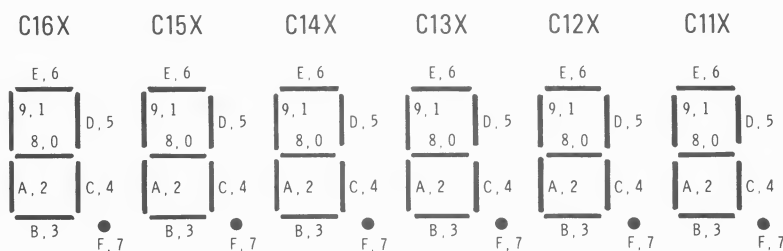


Figure 7-73
Addresses of the various display segments.

Procedure (continued)

16. Load the program shown in Figure 7-74. Verify that you loaded it properly.
17. Execute the program using the DO command. What message does the program write? _____.
18. Notice that each character is written in a different display. Thus, the subroutine OUTCH automatically changes the address to that of the next display after each character is written.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDAA#	Load accumulator A immediate with the
0001	37	37	character byte for the letter H.
0002	BD	JSR	Jump to subroutine
0003	FE	FE	OUTCH
0004	3A	3A	
0005	86	LDAA#	Load ACCA with
0006	4F	4F	next character byte.
0007	BD	JSR	
0008	FE	FE	Display it.
0009	3A	3A	
000A	86	LDAA#	Load next character.
000B	0E	0E	
000C	BD	JSR	
000D	FE	FE	Display it.
000E	3A	3A	
000F	86	LDAA#	Load next character.
0010	67	67	
0011	BD	JSR	
0012	FE	FE	Display it.
0013	3A	3A	
0014	3E	WAI	Stop.

Figure 7-74

This program uses the OUTCH subroutine in the monitor program to display a message.

Discussion

The monitor program writes several messages of its own. Examples are: ACCA, ACCB, CPU UP, and FULL! Thus, the monitor has a subroutine that can be used to write messages. It is called OUTSTR for OUTput a STRing of characters. Its starting address is at FE52₁₆. There is a special convention for calling this subroutine. The JSR FE52₁₆ instruction must be followed immediately by the character bytes that make up the message. Up to six characters can be displayed. The last character must have the decimal point lit. After the message is displayed, control is returned to the instruction immediately following the last character.

Procedure (continued)

19. Load the program shown in Figure 7-75 into the Trainer and verify that you loaded it properly.
20. Execute the program using the DO command. What message does it display? _____.
21. Modify the program so that it displays HELLO.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0000	BD	JSR	Jump to the subroutine that will display the following message.
0001	FE	FE	
0002	52	52	
0003	37	37	H
0004	4F	4F	E
0005	0E	0E	L
0006	E7	E7	P. ← Decimal point must be lit in last character.
0007	3E	WAI	Then stop.

Figure 7-75

The OUTSTR subroutine in the monitor is used to display a message.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	Cal OUTSTR. N O. ← Decimal point lit (last character).
0001	FE	FE	
0002	52	52	
0003	76	76	
0004	FE	FE	
0005	BD	JSR	Call OUTSTR again. G O. ← Decimal point lit (last character). Then stop.
0006	FE	FE	
0007	52	52	
0008	5E	5E	
0009	FE	FE	
000A	3E	WAI	

Figure 7-76
OUTSTR is called twice.

22. The program shown in Figure 7-76 calls the OUTSTR subroutine twice. Load this program into the Trainer.
23. Execute it using the DO command. What message is displayed?
_____.
24. Notice that the second message (GO.) is written to the right of the first. Thus, subroutine OUTSTR does not reset the display to the left for the second message.
25. Rewrite the program so that two blank displays appear between NO. and GO.

Discussion

When displaying long messages such as: "HELLO CAN I HELP YOU?", the display must be given no more than six characters at a time. Also, a short delay must be placed between the various parts of the message. You can achieve a delay by loading the index register with FFFF and decrementing it to 0000. You can achieve an additional delay by using either accumulator in conjunction with the index register. We can write a display subroutine and call it between each part of the message.

Also, because we are using the same displays over again for each part of the message, each new word should start on the left. The subroutine called OUTSTR has an alternate entry point at address FD8C₁₆ called OUTSTJ. The calling convention for this subroutine is the same as that for OUTSTR. However, each new message starts in the left-most display.

Procedure (continued)

26. Load the program shown in Figure 7-77. Verify that you loaded it properly.
27. Execute the program using the DO command. What message is displayed? _____.
28. Change the number in address 003C₁₆, 003E₁₆, and 003F₁₆.
29. Execute the program using the DO command. What affect does this have?
30. Write a program of your own that will display "LOAD 2 IS BAD."

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	
0001	FD	FD	Call OUTSTJ.
0002	8C	8C	
0003	37	37	H
0004	4F	4F	E
0005	0E	0E	L
0006	0E	0E	L
0007	FE	FE	O.
0008	BD	JSR	
0009	00	00	Call Delay Subroutine
000A	2F	2F	
000B	BD	JSR	
000C	FD	FD	Call OUTSTJ again.
000D	8C	8C	
000E	4E	4E	C
000F	77	77	A
0010	76	76	N
0011	00	00	blank
0012	B0	B0	I.
0013	BD	JSR.	
0014	00	00	Call Delay Subroutine
0015	2F	2F	
0016	BD	JSR	
0017	FD	FD	Call OUTSTJ again.
0018	8C	8C	
0019	37	37	H
001A	4F	4F	E
001B	0E	0E	L
001C	67	67	P
001D	80	80	•
001E	BD	JSR	Call Delay Subroutine
001F	00	00	
0020	2F	2F	
0021	BD	JSR	Call OUTSTJ again.
0022	FD	FD	
0023	8C	8C	
0024	3B	3B	Y
0025	7E	7E	O
0026	3E	3E	U
0027	00	00	blank
0028	80	80	•
0029	BD	JSR	
002A	00	00	
002B	2F	2F	
002C	7E	JMP	Do it all again.
002D	00	00	
002E	00	00	
002F	CE	LDX#	
0030	FF	FF	} Delay subroutine.
0031	FF	FF	
0032	09	DEX	
0033	26	BNE	
0034	FD	FD	
0035	39	RTS	

Figure 7-77

This program makes extensive use of the subroutine call.

Discussion

The monitor program in the Trainer contains some other useful subroutines. These are outlined in the manual for the ET-6800 Microprocessor Trainer. Two of the most useful are REDIS and OUTBYT.

OUTBYT is a subroutine that displays the contents of accumulator A as two hex digits. Its address is $FE20_{16}$. When this subroutine is called for the first time, the two left displays are used. If it is called again without being reset, the two center displays are used. The third time, the two right displays are used.

The display can be reset to the left by calling the REDIS subroutine. This subroutine is located in address $FCBC_{16}$. If OUTBYT is called after REDIS is called, the two left displays will be used.

Procedure (continued)

31. Load the program shown in Figure 7-78. Verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear accumulator A
0001	BD	JSR	
0002	FE	FE	Call OUTBYT
0003	20	20	
0004	BD	JSR	
0005	00	00	Call Delay Subroutine
0006	0E	0E	
0007	4C	INCA	Increment accumulator A
0008	BD	JSR	
0009	FC	FC	Call REDIS
000A	BC	BC	
000B	7E	JMP	
000C	00	00	Do it again.
000D	01	01	
000E	CE	LDX#	
000F	FF	FF	
0010	FF	FF	
0011	09	DEX	} Delay Subroutine.
0012	26	BNE	
0013	FD	FD	
0014	39	RTS	

Figure 7-78

This routine counts seconds from 00 to 99.

32. Execute the program using the DO command.
33. Which digits are used by the display? _____.
34. Notice that the JSR instruction at address 0008 calls the subroutine that resets the display to the left.
35. To illustrate why this is necessary, let's see what happens when this important step is omitted. Change the contents of locations 0008, 0009, and 000A to 01. This replaces the JSR instruction with three NOPs.

36. Execute the program using the DO command. Notice that, without calling the REDIS subroutine, the display advances to the right and is lost after the third time through the loop.
37. Restore the program to its original state. How can the count be speeded up?

Discussion

The speed of the count can be varied by changing the contents of addresses 000F and 0010. It probably has occurred to you that the trainer could be turned into a digital clock. In the following procedure, you will develop a program that will do this.

Procedure

38. Write a program that will count seconds from 00 to 99₁₀. The seconds count should be maintained in the two left-most displays. It should count as the above program did, but in decimal instead of hexadecimal.
39. If you have problems, remember that the DAA instruction can be used to convert the addition of BCD numbers to a BCD sum. However, the DAA instruction works only if preceded immediately by an ADDA or ADCA instruction.
40. Load your program into the Trainer and execute it using the DO command.

Discussion

One solution is shown in Figure 7-79. Carefully study this program. This routine counts the seconds in decimal. However in a real digital clock, the seconds reset to 00 after 59_{10} rather than after 99_{10} .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear seconds.
0001	BD	JSR	
0002	FE	FE	Call OUTBYT
0003	20	20	
0004	BD	JSR	
0005	00	00	Call Delay subroutine
0006	10	10	
0007	8B	ADDA#	Increment seconds
0008	01	01	
0009	19	DAA	Make it decimal
000A	BD	JSR	
000B	FC	FC	Call REDIS
000C	BC	BC	
000D	7E	JMP	
000E	00	00	Do it all again.
000F	01	01	
0010	CE	LDX#	} One second Delay Subroutine
0011	B4	B4	
0012	00	00	
0013	09	DEX	
0014	26	BNE	
0015	FD	FD	
0016	CE	LOX#	
0017	FF	FF	
0018	FF	FF	
0019	09	DEX	
001A	26	BNE	
001B	FD	FD	
001C	39	RTS	

Figure 7-79

This routine counts seconds from 00 to 99.

Procedure (continued)

41. Modify your program (or the one in this Experiment) so that it displays seconds from 00 to 59 and then returns to 00 and starts over again.
42. Load your program into the Trainer and execute it using the DO command.
43. Debug your program if necessary until it performs properly.

Discussion

One solution is shown in Figure 7-80. The seconds count is compared to 60 each time it is incremented. When it reaches 60, it is reset to 00.

The next step is to add a minutes count. This can be done by incrementing a decimal number each time the seconds count “rolls over” from 59 to 00. The decimal number is then displayed as minutes.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	C6	LDAB#	Load number for comparison
0001	60	60	
0002	4F	CLRA	Clear seconds.
0003	BD	JSR	
0004	FE	FE	Call OUTBYT
0005	20	20	
0006	BD	JSR	
0007	00	00	Call Delay Subroutine
0008	14	14	
0009	BD	JSR	
000A	FC	FC	Call REDIS
000B	BC	BC	
000C	8B	ADDA#	Increment seconds.
000D	01	01	
000E	19	DAA	Make it decimal
000F	11	CBA	Time to clear seconds
0010	27	BEQ	Yes.
0011	F0	F0	
0012	20	BRA	No.
0013	EF	EF	
0014	CE	LDX#	
0015	B4	B4	
0016	00	00	} One second Delay Subroutine
0017	09	DEX	
0018	26	BNE	
0019	FD	FD	
001A	CE	LDX#	
001B	FF	FF	
001C	FF	FF	
001D	09	DEX	
001E	26	BNE	
001F	FD	FD	
0020	39	RTS	

Figure 7-80

This routine counts seconds from 00 to 59.

Procedure (continued)

44. Write a program that will display minutes and seconds properly. The minutes should be displayed in the two left displays; the seconds in the two center displays. Like the seconds, the minutes should return to 00 after 59.
45. Load your program and execute it.
46. Debug your program as necessary.

Discussion

A solution is shown in Figure 7-81. Your approach may be more straightforward, but may require more memory.

The final step is to include the hours display.

Procedure (continued)

47. Modify your program so that it displays hours, minutes and seconds.
48. Load your program and execute it.
49. Debug your program as necessary.

A solution is shown in Figure 7-82. This program evolved over a period of time and is extremely compact. It is virtually impossible for a beginning programmer to write a program this compact on the first try. Your program may require substantially more memory, but the important thing is: does it work?

You can "fine tune" the clock period by changing the numbers in addresses 0004 and 0005. The clock will be fairly accurate because its timing accuracy is derived from the quartz crystal MPU clock in your trainer.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	00	00	Reserved for seconds
0001	00	00	Reserved for minutes
0002	CE	LDX#	} One second delay.
0003	B4	B4	
0004	00	00	
0005	09	DEX	
0006	26	BNE	
0007	FD	FD	
0008	CE	LDX#	
0009	FF	FF	
000A	FF	FF	
000B	09	DEX	
000C	26	BNE	
000D	FD	FD	
000E	C6	LDAB#	Load number for comparison
000F	60	60	
0010	0D	SEC	Set carry bit.
0011	8D	BSR	Branch to subroutine to increment seconds.
0012	11	11	
0013	8D	BSR	Branch to the same subroutine to increment minutes
0014	0F	0F	
0015	BD	JSR	
0016	FC	FC	Call REDIS
0017	BC	BC	
0018	96	LDAA	Load minutes
0019	01	01	
001A	BD	JSR	
001B	FE	FE	Call OUTBYT to display minutes
001C	20	20	
001D	96	LDAA	Load seconds
001E	00	00	
001F	BD	JSR	Call OUTBYT to display seconds
0020	FE	FE	
0021	20	20	
0022	20	BRA	Do it all again.
0023	DE	DE	
0024	A6	LDAA,X	Load seconds (or minutes) into A.
0025	00	00	
0026	89	ADCA#	Increment if necessary
0027	00	00	
0028	19	DAA	Adjust to decimal
0029	11	CBA	Time to clear?
002A	26	BNE	No.
002B	01	01	
002C	4F	CLRA	Yes.
002D	A7	STAA,X	Store seconds (or minutes)
002E	00	00	
002F	08	INX	
0030	07	TPA	
0031	88	EORA#	Complement carry bit
0032	01	01	
0033	06	TAP	
0034	39	RTS	

Increment subroutine

Figure 7-81
Routine for displaying minutes and seconds.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	00	00	Reserved for seconds
0001	00	00	Reserved for minutes
0002	00	00	Reserved for hours
0003	CE	LDX#	} One second Delay
0004	B4	B4	
0005	00	00	
0006	09	DEX	
0007	26	BNE	
0008	FD	FD	
0009	CE	LDX#	
000A	FF	FF	
000B	FF	FF	
000C	09	DEX	
000D	26	BNE	} Minutes and seconds will be compared with sixty Prepare to increment seconds Go to subroutine that will increment seconds. Go to same subroutine. It will increment Minutes if necessary Hours will be compared with twelve Go to same subroutine. It will increment hours if necessary.
000E	FD	FD	
000F	C6	LDAB#	
0010	60	60	
0011	0D	SEC	
0012	8D	BSR	
0013	11	11	
0014	8D	BSR	
0015	0F	0F	
0016	C6	LDAB#	
0017	12	12	} Call REDIS Call display subroutine to display hours. Call display subroutine to display minutes. Call display subroutine to display seconds. Do it all again.
0018	8D	BSR	
0019	0B	0B	
001A	BD	JSR	
001B	FC	FC	
001C	BC	BC	
001D	8D	BSR	
001E	17	17	
001F	8D	BSR	
0020	15	15	
0021	8D	BSR	
0022	13	13	
0023	20	BRA	
0024	DE	DE	

cont'd.-

cont'd.			
0025	A6	LDAA,X	Load seconds (or minutes or hours).
0026	00	00	
0027	89	ADCA#	Increment if necessary.
0028	00	00	
0029	19	DAA	Adjust to decimal
002A	11	CBA	Time to clear?
002B	25	BCS	No.
002C	01	01	
002D	4F	CLRA	Yes.
002E	A7	STAA,X	Store seconds (or minutes or hours).
002F	00	00	
0030	08	INX	Point index register at minutes (or hours).
0031	07	TPA	
0032	88	EORA#	Complement carry bit
0033	01	01	
0034	06	TAP	
0035	39	RTS	
0036	09	DEX	Point index register at hours (or minutes or seconds)
0037	A6	LDAA,X	Load hours (or minutes or seconds)
0038	00	00	
0039	7E	JSR	Display hours (or minutes or seconds)
003A	FE	FE	
003B	20	20	
003C	39	RTS	

Increment Subroutine

Display Subroutine

Figure 7-82
Twelve-hour clock program.

Appendix A

DEFINITION OF THE EXECUTABLE INSTRUCTIONS

A.1 Nomenclature

The following nomenclature is used in the subsequent definitions.

(a) Operators

()	= contents of
←	= is transferred to
↑	= "is pulled from stack"
↓	= "is pushed into stack"
·	= Boolean AND
⊙	= Boolean (Inclusive) OR
⊕	= Exclusive OR
≈	= Boolean NOT

(b) Registers in the MPU

ACCA	= Accumulator A
ACCB	= Accumulator B
ACCX	= Accumulator ACCA or ACCB
CC	= Condition codes register
IX	= Index register, 16 bits
IXH	= Index register, higher order 8 bits
IXL	= Index register, lower order 8 bits
PC	= Program counter, 16 bits
PCH	= Program counter, higher order 8 bits
PCL	= Program counter, lower order 8 bits
SP	= Stack pointer
SPH	= Stack pointer high
SPL	= Stack pointer low

(c) Memory and Addressing

M	= A memory location (one byte)
M + 1	= The byte of memory at 0001 plus the address of the memory location indicated by "M."
Rel	= Relative address (i.e. the two's complement number stored in the second byte of machine code corresponding to a branch instruction).

(d) Bits 0 thru 5 of the Condition Codes Register

C	= Carry — borrow	bit — 0
V	= Two's complement overflow indicator	bit — 1
Z	= Zero indicator	bit — 2
N	= Negative indicator	bit — 3
I	= Interrupt mask	bit — 4
H	= Half carry	bit — 5

(e) Status of Individual Bits BEFORE Execution of an Instruction

An	= Bit n of ACCA (n=7,6,5,...,0)
Bn	= Bit n of ACCB (n=7,6,5,...,0)
IXHn	= Bit n of IXH (n=7,6,5,...,0)

IXLn = Bit n of IXL (n=7,6,5,...,0)

Mn = Bit n of M (n=7,6,5,...,0)

SPHn = Bit n of SPH (n=7,6,5,...,0)

SPLn = Bit n of SPL (n=7,6,5,...,0)

Xn = Bit n of ACCX (n=7,6,5,...,0)

(f) *Status of Individual Bits of the RESULT of Execution of an Instruction*

(i) For 8-bit Results

Rn = Bit n of the result (n = 7,6,5,...,0)

This applies to instructions which provide a result contained in a single byte of memory or in an 8-bit register.

(ii) For 16-bit Results

RHn = Bit n of the more significant byte of the result
(n = 7,6,5,...,0)

RLn = Bit n of the less significant byte of the result
(n = 7,6,5,...,0)

This applies to instructions which provide a result contained in two consecutive bytes of memory or in a 16-bit register.

A.2 Executable Instructions (definition of)

Detailed definitions of the 72 executable instructions of the source language are provided on the following pages.

Add Accumulator B to Accumulator A**ABA**

Operation: $ACCA \leftarrow (ACCA) + (ACCB)$

Description: Adds the contents of ACCB to the contents of ACCA and places the result in ACCA.

Condition Codes: H: Set if there was a carry from bit 3; cleared otherwise.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = A_3 \cdot B_3 + B_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot A_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot B_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot \bar{B}_7 \cdot R_7$$

$$C = A_7 \cdot B_7 + B_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot A_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
Inherent	2	1	1B	033	027

ADC

Add with Carry

Operation: $ACCX \leftarrow (ACCX) + (M) + (C)$

Description: Adds the contents of the C bit to the sum of the contents of ACCX and M, and places the result in ACCX.

Condition Codes: H Set if there was a carry from bit 3; cleared otherwise.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot X_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot M_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot \bar{M}_7 \cdot R_7$$

$$C = X_7 \cdot M_7 + M_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot X_7$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	89	211	137
A DIR	3	2	99	231	153
A EXT	4	3	B9	271	185
A IND	5	2	A9	251	169
B IMM	2	2	C9	311	201
B DIR	3	2	D9	331	217
B EXT	4	3	F9	371	249
B IND	5	2	E9	351	233

Add Without Carry**ADD**

Operation: $ACCX \leftarrow (ACCX) + (M)$

Description: Adds the contents of ACCX and the contents of M and places the result in ACCX.

Condition Codes: H: Set if there was a carry from bit 3; cleared otherwise.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot X_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot M_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot \bar{M}_7 \cdot R_7$$

$$C = X_7 \cdot M_7 + M_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot X_7$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8B	213	139
A DIR	3	2	9B	233	155
A EXT	4	3	BB	273	187
A IND	5	2	AB	253	171
B IMM	2	2	CB	313	203
B DIR	3	2	DB	333	219
B EXT	4	3	FB	373	251
B IND	5	2	EB	353	235

AND

Logical AND

Operation: $ACCX \leftarrow (ACCX) \cdot (M)$

Description: Performs logical "AND" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "AND" of the corresponding bits of M and of ACCX before the operation.)

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

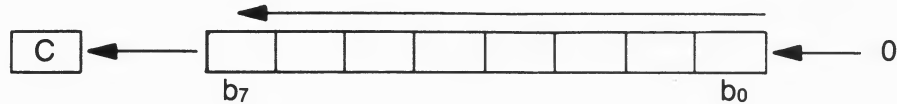
See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	84	204	132
A DIR	3	2	94	224	148
A EXT	4	3	B4	264	180
A IND	5	2	A4	244	164
B IMM	2	2	C4	304	196
B DIR	3	2	D4	324	212
B EXT	4	3	F4	364	244
B IND	5	2	E4	344	228

Arithmetic Shift Left**ASL**

Operation:



Description: Shifts all bits of the ACCX or M one place to the left. Bit 0 is loaded with a zero. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
 C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation)

$$C = M_7$$

Addressing Formats

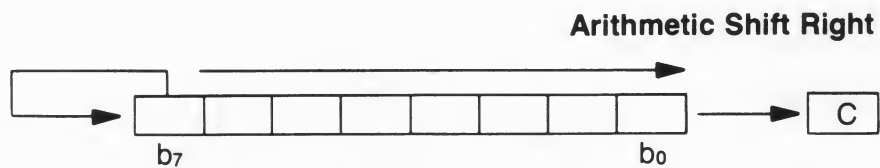
See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	48	110	072
B	2	1	58	130	088
EXT	6	3	78	170	120
IND	7	2	68	150	104

ASR

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation)

$$C = M_0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	47	107	071
B	2	1	57	127	087
EXT	6	3	77	167	119
IND	7	2	67	147	103

Branch if Carry Clear**BCC**

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(C)=0$

Description: Tests the state of the C bit and causes a branch if C is clear.
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	24	044	036

BCS

Branch if Carry Set

Operation: $PC \leftarrow (PC) + 0002 + \text{Rel if } (C)=1$

Description: Tests the state of the C bit and causes a branch if C is set.
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	25	045	037

Branch if Equal**BEQ**

Operation: $PC \leftarrow (PC) + 0002 + \text{Rel if } (Z)=1$

Description: Tests the state of the Z bit and causes a branch if the Z bit is set.
See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	27	047	039

BGE

Branch if Greater than or Equal to Zero

Operation: $PC \leftarrow (PC) + 0002 + \text{Rel if } (N) \oplus (V) = 0$
i.e. if $(ACCX) \geq (M)$
(Two's complement numbers)

Description: Causes a branch if (N is set and V is set) OR (N is clear and V is clear).
If the BGE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than or equal to the two's complement number represented by the subtrahend (i.e. M).
See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2C	054	044

Branch if Greater than Zero**BGT**

Operation: $PC \leftarrow (PC) + 0002 + \text{Rel if } (Z) \ominus [(N) \oplus (V)] = 0$

i.e. if $(ACCX) > (M)$
(two's complement numbers)

Description: Causes a branch if [Z is clear] AND [(N is set and V is set) OR (N is clear and V is clear)].

If the BGT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2E	056	046

BHI

Branch if Higher

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(C) \cdot (Z)=0$
i.e. if $(ACCX) > (M)$
(unsigned binary numbers)

Description:	<p>Causes a branch if (C is clear) AND (Z is clear).</p> <p>If the BHI instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was greater than the unsigned binary number represented by the subtrahend (i.e. M).</p>
--------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	22	042	034

Bit Test**BIT**

Operation: $(ACCX) \cdot (M)$

Description: Performs the logical "AND" comparison of the contents of ACCX and the contents of M and modifies condition codes accordingly. Neither the contents of ACCX or M operands are affected. (Each bit of the result of the "AND" would be the logical "AND" of the corresponding bits of M and ACCX.)

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the result of the "AND" would be set; cleared otherwise.
 Z: Set if all bits of the result of the "AND" would be cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	85	205	133
A DIR	3	2	95	225	149
A EXT	4	3	B5	265	181
A IND	5	2	A5	245	165
B IMM	2	2	C5	305	197
B DIR	3	2	D5	325	213
B EXT	4	3	F5	365	245
B IND	5	2	E5	345	229

BLE

Branch if Less than or Equal to Zero

Operation: $PC \leftarrow (PC) + 0002 + Rel \text{ if } (Z) \odot [(N) \oplus (V)] = 1$

i.e. if $(ACCX) \leq (M)$

(two's complement numbers)

Description: Causes a branch if [Z is set] OR [(N is set and V is clear) OR (N is clear and V is set)].

If the BLE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2F	057	047

BLS

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	23	043	035

BLT

Branch if Less than Zero

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(N) \oplus (V) = 1$
 i.e. if $(ACCX) < (M)$
 (two's complement numbers)

Description: Causes a branch if (N is set and V is clear) OR (N is clear and V is set).
 If the BLT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2D	055	045

Branch if Minus**BMI**

Operation: $PC \leftarrow (PC) + 0002 + \text{Rel if } (N) = 1$

Description: Tests the state of the N bit and causes a branch if N is set.
See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2B	053	043

BNE

Branch if Not Equal

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(Z) = 0$

Description: Tests the state of the Z bit and causes a branch if the Z bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	26	046	038

Branch if Plus**BPL**

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if (N) = 0

Description: Tests the state of the N bit and causes a branch if N is clear.
See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2A	052	042

BRA**Branch Always**

Operation: $PC \leftarrow (PC) + 0002 + Rel$

Description: Unconditional branch to the address given by the foregoing formula, in which R is the relative address stored as a two's complement number in the second byte of machine code corresponding to the branch instruction.

Note: The source program specifies the destination of any branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler obtains the relative address R from the absolute address and the current value of the program counter PC.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	20	040	032

Branch to Subroutine**BSR**

Operation: $PC \leftarrow (PC) + 0002$
 $\downarrow (PCL)$
 $SP \leftarrow (SP) - 0001$
 $\downarrow (PCH)$
 $SP \leftarrow (SP) - 0001$
 $PC \leftarrow (PC) + Rel$

Description: The program counter is incremented by 2. The less significant byte of the contents of the program counter is pushed into the stack. The stack pointer is then decremented (by 1). The more significant byte of the contents of the program counter is then pushed into the stack. The stack pointer is again decremented (by 1). A branch then occurs to the location specified by the program.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	8	2	8D	215	141

BRANCH TO SUBROUTINE EXAMPLE

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A.	Before					
	PC	\leftarrow \$1000	8D		BSR	CHARLI
		\$1001	50			
	SP	\leftarrow \$EFFF				
B.	After					
	PC	\leftarrow \$1052	**	CHARLI	***	*****
	SP	\leftarrow \$EFFD				
		\$EFFE	10			
		\$EFFF	02			

BVC

Branch if Overflow Clear

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(V) = 0$

Description: Tests the state of the V bit and causes a branch if the V bit is clear.
See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	28	050	040

Branch if Overflow Set**BVS**

Operation: $PC \leftarrow (PC) + 0002 + Rel$ if $(V) = 1$

Description: Tests the state of the V bit and causes a branch if the V bit is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	29	051	041

CBA

Compare Accumulators

Operation: (ACCA) – (ACCB)

Description: Compares the contents of ACCA and the contents of ACCB and sets the condition codes, which may be used for arithmetic and logical conditional branches. Both operands are unaffected.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
- Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
- V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
- C: Set if the subtraction would require a borrow into the most significant bit of the result; clear otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot \bar{B}_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot B_7 \cdot R_7$$

$$C = \bar{A}_7 \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \bar{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	11	021	017

Clear Carry**CLC**

Operation: C bit \leftarrow 0

Description: Clears the carry bit in the processor condition codes register.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Not affected.
 Z: Not affected.
 V: Not affected.
 C: Cleared

Boolean Formulae for Condition Codes:
 C = 0

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0C	014	012

CLI**Clear Interrupt Mask**

Operation: $I \text{ bit} \leftarrow 0$

Description: Clears the interrupt mask bit in the processor condition codes register. This enables the microprocessor to service an interrupt from a peripheral device if signalled by a high state of the "Interrupt Request" control input.

Condition Codes: H: Not affected.
 I: Cleared.
 N: Not affected.
 Z: Not affected.
 V: Not affected.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$I = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0E	016	014

Clear**CLR**

Operation: $ACCX \leftarrow 00$

or: $M \leftarrow 00$

Description: The contents of ACCX or M are replaced with zeros.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Cleared
 Z: Set
 V: Cleared
 C: Cleared

Boolean Formulae for Condition Codes:

$N = 0$

$Z = 1$

$V = 0$

$C = 0$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4F	117	079
B	2	1	5F	137	095
EXT	6	3	7F	177	127
IND	7	2	6F	157	111

CLV**Clear Two's Complement Overflow Bit**

Operation: $V \text{ bit} \leftarrow 0$

Description: Clears the two's complement overflow bit in the processor condition codes register.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Not affected.
 Z: Not affected.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:
 $V = 0$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0A	012	010

Compare**CMP**

Operation: (ACCX) – (M)

Description: Compares the contents of ACCX and the contents of M and determines the condition codes, which may be used subsequently for controlling conditional branching. Both operands are unaffected.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
 Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
 V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
 C: Carry is set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	81	201	129
A DIR	3	2	91	221	145
A EXT	4	3	B1	261	177
A IND	5	2	A1	241	161
B IMM	2	2	C1	301	193
B DIR	3	2	D1	321	209
B EXT	4	3	F1	361	241
B IND	5	2	E1	341	225

COM

Complement

Operation: $ACCX \leftarrow \approx (ACCX) = FF - (ACCX)$

or: $M \leftarrow \approx (M) = FF - (M)$

Description: Replaces the contents of ACCX or M with its one's complement. (Each bit of the contents of ACCX or M is replaced with the complement of that bit.)

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Cleared.

C: Set.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

$$C = 1$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	43	103	067
B	2	1	53	123	083
EXT	6	3	73	163	115
IND	7	2	63	143	099

Compare Index Register**CPX**

Operation: $(IXL) - (M+1)$
 $(IXH) - (M)$

Description: The more significant byte of the contents of the index register is compared with the contents of the byte of memory at the address specified by the program. The less significant byte of the contents of the index register is compared with the contents of the next byte of memory, at one plus the address specified by the program. The Z bit is set or reset according to the results of these comparisons, and may be used subsequently for conditional branching.

The N and V bits, though determined by this operation, are not intended for conditional branching.

The C bit is not affected by this operation.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the result of the subtraction from the more significant byte of the index register would be set; cleared otherwise.
 Z: Set if all bits of the results of both subtractions would be cleared; cleared otherwise.
 V: Set if the subtraction from the more significant byte of the index register would cause two's complement overflow; cleared otherwise.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = IXH_7 \cdot M_7 \cdot RH_7 + IXH_7 \cdot M_7 \cdot RH_7$$

Addressing Formats:

See Table A-5 (Page A-80).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8C	214	140
DIR	4	2	9C	234	156
EXT	5	3	BC	274	188
IND	6	2	AC	254	172

DAA**Decimal Adjust ACCA**

Operation: Adds hexadecimal numbers 00, 06, 60, or 66 to ACCA, and may also set the carry bit, as indicated in the following table:

State of C-bit before DAA (Col. 1)	Upper Half-byte (bits 4-7) (Col. 2)	Initial Half-carry H-bit (Col.3)	Lower to ACCA (bits 0-3) (Col. 4)	Number Added after by DAA (Col. 5)	State of C-bit DAA (Col. 6)
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-3	1	0-3	66	1

Note: Columns (1) through (4) of the above table represent all possible cases which can result from any of the operations ABA, ADD, or ADC, with initial carry either set or clear, applied to two binary-coded-decimal operands. The table shows hexadecimal values.

Description: If the contents of ACCA and the state of the carry-borrow bit C and the half-carry bit H are all the result of applying any of the operations ABA, ADD, or ADC to binary-coded-decimal operands, with or without an initial carry, the DAA operation will function as follows.

Subject to the above condition, the DAA operation will adjust the contents of ACCA and the C bit to represent the correct binary-coded-decimal sum and the correct state of the carry.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Not defined.
 C: Set or reset according to the same rule as if the DAA and an immediately preceding ABA, ADD, or ADC were replaced by a hypothetical binary-coded-decimal addition.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

C = See table above.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	19	031	025

DEC**Decrement**

Operation: $ACCX \leftarrow (ACCX) - 01$

or: $M \leftarrow (M) - 01$

Description: Subtract one from the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (ACCX) or (M) was 80 before the operation.

C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{X}_6 \cdot \bar{X}_5 \cdot \bar{X}_4 \cdot \bar{X}_3 \cdot \bar{X}_2 \cdot \bar{X}_0 = \bar{R}_7 \cdot R_6 \cdot R_5 \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \cdot R_0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4A	112	074
B	2	1	5A	132	090
EXT	6	3	7A	172	122
IND	7	2	6A	152	106

Decrement Stack Pointer**DES**Operation: $SP \leftarrow (SP) - 0001$

Description: Subtract one from the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	34	064	052

DEX

Decrement Index Register

Operation: $IX \leftarrow (IX) - 0001$

Description: Subtract one from the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Not affected.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Not affected.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	09	011	009

Exclusive OR**EOR**

Operation: $ACCX \leftarrow (ACCX) \oplus (M)$

Description: Perform logical "EXCLUSIVE OR" between the contents of ACCX and the contents of M, and place the result in ACCX. (Each bit of ACCX after the operation will be the logical "EXCLUSIVE OR" of the corresponding bit of M and ACCX before the operation.)

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Cleared
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	88	210	136
A DIR	3	2	98	230	152
A EXT	4	3	B8	270	184
A IND	5	2	A8	250	168
B IMM	2	2	C8	310	200
B DIR	3	2	D8	330	216
B EXT	4	3	F8	370	248
B IND	5	2	E8	350	232

INC**Increment**

Operation: $ACCX \leftarrow (ACCX) + 01$

or: $M \leftarrow (M) + 01$

Description: Add one to the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow will occur if and only if (ACCX) or (M) was 7F before the operation.

C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = \bar{X}_7 \cdot X_6 \cdot X_5 \cdot X_4 \cdot X_3 \cdot X_2 \cdot X_1 \cdot X_0$$

$$C = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4C	114	076
B	2	1	5C	134	092
EXT	6	3	7C	174	124
IND	7	2	6C	154	108

Increment Stack Pointer**INS**Operation: $SP \leftarrow (SP) + 0001$

Description: Add one to the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	31	061	049

INX

Increment Index Register

Operation: $IX \leftarrow (IX) + 0001$

Description: Add one to the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Not affected.
 Z: Set if all 16 bits of the result are cleared; cleared otherwise.
 V: Not affected.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	08	010	008

Jump**JMP**

Operation: PC ← numerical address

Description: A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table A-7 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	3	3	7E	176	126
IND	4	2	6E	156	110

JSR

Jump to Subroutine

Operation:

Either: $PC \leftarrow (PC) + 0003$ (for EXTended addressing)

or: $PC \leftarrow (PC) + 0002$ (for INDexed addressing)

Then: $\downarrow (PCL)$

$SP \leftarrow (SP) - 0001$

$\downarrow (PCH)$

$SP \leftarrow (SP) - 0001$

$PC \leftarrow$ numerical address

Description: The program counter is incremented by 3 or by 2, depending on the addressing mode, and is then pushed onto the stack, eight bits at a time. The stack pointer points to the next empty location in the stack. A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table A-7 (Page A-82).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	9	3	BD	275	189
IND	8	2	AD	255	173

JUMP TO SUBROUTINE EXAMPLE (extended mode)

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before:						
PC	→	\$0FFF	BD		JSR	CHARLI
		\$1000	20			
		\$1001	77			
SP	←	\$EFFF				
B. After:						
PC	→	\$2077	**	CHARLI	***	*****
SP	→	\$EFFD				
		\$EFFE	10			
		\$EFFF	02			

Load Accumulator**LDA**

Operation: $ACCX \leftarrow (M)$

Description: Loads the contents of memory into the accumulator. The condition codes are set according to the data.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	86	206	134
A DIR	3	2	96	226	150
A EXT	4	3	B6	266	182
A IND	5	2	A6	246	166
B IMM	2	2	C6	306	198
B DIR	3	2	D6	326	214
B EXT	4	3	F6	366	246
B IND	5	2	E6	346	230

LDS**Load Stack Pointer**

Operation: $SPH \leftarrow (M)$
 $SPL \leftarrow (M+1)$

Description: Loads the more significant byte of the stack pointer from the byte of memory at the address specified by the program, and loads the less significant byte of the stack pointer from the next byte of memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the stack pointer is set by the operation; cleared otherwise.
 Z: Set if all bits of the stack pointer are cleared by the operation; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-5 (Page A-80).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8E	216	142
DIR	4	2	9E	236	158
EXT	5	3	BE	276	190
IND	6	2	AE	256	174

Load Index Register**LDX**

Operation: $IXH \leftarrow (M)$
 $IXL \leftarrow (M+1)$

Description: Loads the more significant byte of the index register from the byte of memory at the address specified by the program, and loads the less significant byte of the index register from the next byte of memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the index register is set by the operation; cleared otherwise.
 Z: Set if all bits of the index register are cleared by the operation; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-5 (Page A-80).

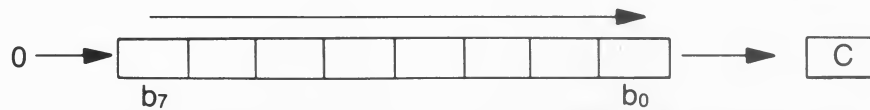
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	CE	316	206
DIR	4	2	DE	336	222
EXT	5	3	FE	376	254
IND	6	2	EE	356	238

LSR

Logical Shift Right

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded with a zero. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Cleared.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
 C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = 0$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation).

$$C = M_0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	44	104	068
B	2	1	54	124	084
EXT	6	3	74	164	116
IND	7	2	64	144	100

Negate**NEG**

Operation: $ACCX \leftarrow - (ACCX) = 00 - (ACCX)$

or: $M \leftarrow - (M) = 00 - (M)$

Description: Replaces the contents of ACCX or M with its two's complement. Note that 80 is left unchanged.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there would be two's complement overflow as a result of the implied subtraction from zero; this will occur if and only if the contents of ACCX or M is 80.
 C: Set if there would be a borrow in the implied subtraction from zero; the C bit will be set in all cases except when the contents of ACCX or M is 00.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = R_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$C = R_7 + R_6 + R_5 + R_4 + R_3 + R_2 + R_1 + R_0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	40	100	064
B	2	1	50	120	080
EXT	6	3	70	160	112
IND	7	2	60	140	096

NOP

No Operation

Description: This is a single-word instruction which causes only the program counter to be incremented. No other registers are affected.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	01	001	001

Inclusive OR**ORA**

Operation: $ACCX \leftarrow (ACCX) \odot (M)$

Description: Perform logical "OR" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "OR" of the corresponding bits of M and of ACCX before the operation).

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8A	212	138
A DIR	3	2	9A	232	154
A EXT	4	3	BA	272	186
A IND	5	2	AA	252	170
B IMM	2	2	CA	312	202
B DIR	3	2	DA	332	218
B EXT	4	3	FA	372	250
B IND	5	2	EA	352	234

PSH

Push Data Onto Stack

Operation: ↓ (ACCX)
 $SP \leftarrow (SP) - 0001$

Description: The contents of ACCX is stored in the stack at the address contained in the stack pointer. The stack pointer is then decremented.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4 (Page A-80).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	36	066	054
B	4	1	37	067	055

Pull Data from Stack**PUL**

Operation: $SP \leftarrow (SP) + 0001$
 $\uparrow ACCX$

Description: The stack pointer is incremented. The ACCX is then loaded from the stack, from the address which is contained in the stack pointer.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4 (Page A-80).

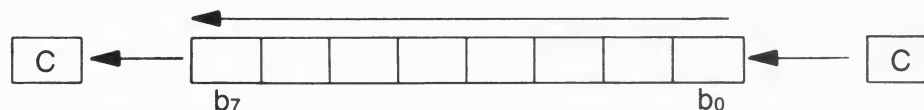
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	32	062	050
B	4	1	33	063	051

ROL

Rotate Left

Operation:



Description: Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the rotation)

$$C = M_7$$

Addressing Formats:

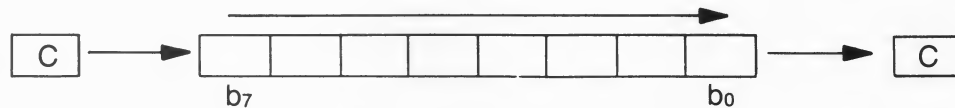
See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	49	111	073
B	2	1	59	131	089
EXT	6	3	79	171	121
IND	7	2	69	151	105

Rotate Right**ROR**

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded from the C bit. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
 C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the rotation)

$$C = M_0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	46	106	070
B	2	1	56	126	086
EXT	6	3	76	166	118
IND	7	2	66	146	102

RTI**Return from Interrupt**

Operation: $SP \leftarrow (SP) + 0001, \uparrow CC$
 $SP \leftarrow (SP) + 0001, \uparrow ACCB$
 $SP \leftarrow (SP) + 0001, \uparrow ACCA$
 $SP \leftarrow (SP) + 0001, \uparrow IXH$
 $SP \leftarrow (SP) + 0001, \uparrow IXL$
 $SP \leftarrow (SP) + 0001, \uparrow PCH$
 $SP \leftarrow (SP) + 0001, \uparrow PCL$

Description: The condition codes, accumulators B and A, the index register, and the program counter, will be restored to a state pulled from the stack. Note that the interrupt mask bit will be reset if and only if the corresponding bit stored in the stack is zero.

Condition Codes: Restored to the states pulled from the stack.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	10	1	3B	073	059

Return from Interrupt

Example

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A.	Before					
	PC	→ \$D066	3B		RTI	
	SP	→ \$EFFF				
		\$EFFF	11HINZVC	(binary)		
		\$EFFF	12			
		\$EFFF	34			
		\$EFFF	56			
		\$EFFF	78			
		\$EFFF	55			
		\$EFFF	67			
B.	After					
	PC	→ \$5567	**		***	*****
		\$EFFF				
		\$EFFF	11HINZVC	(binary)		
		\$EFFF	12			
		\$EFFF	34			
		\$EFFF	56			
		\$EFFF	78			
		\$EFFF	55			
	SP	→ \$EFFF	67			

CC = HINZVC (binary)

ACCB = 12 (Hex)

IXH = 56 (Hex)

ACCA = 34 (Hex)

IXL = 78 (Hex)

Return from Subroutine**RTS**

Operation: $SP \leftarrow (SP) + 0001$
 $\uparrow PCH$
 $SP \leftarrow (SP) + 0001$
 $\uparrow PCL$

Description: The stack pointer is incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of highest significance in the program counter. The stack pointer is again incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of lowest significance in the program counter.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	5	1	39	071	057

Return from Subroutine**EXAMPLE**

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A.	Before					
	PC	\$30A2	39		RTS	
	SP	\$EFFF				
		\$EFFF	10			
		\$EFFF	02			
B.	After					
	PC	\$1002	**		***	*****
		\$EFFF				
		\$EFFF	10			
	SP	\$EFFF	02			

SBA**Subtract Accumulators**

Operation: $ACCA \leftarrow (ACCA) - (ACCB)$

Description: Subtracts the contents of ACCB from the contents of ACCA and places the result in ACCA. The contents of ACCB are not affected.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there was two's complement overflow as a result of the operation.
 C: Carry is set if the absolute value of accumulator B plus previous carry is larger than the absolute value of accumulator A; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \overline{R_7} \cdot \overline{R_6} \cdot \overline{R_5} \cdot \overline{R_4} \cdot \overline{R_3} \cdot \overline{R_2} \cdot \overline{R_1} \cdot \overline{R_0}$$

$$V = A_7 \cdot \overline{B_7} \cdot \overline{R_7} + \overline{A_7} \cdot B_7 \cdot R_7$$

$$C = \overline{A_7} \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \overline{A_7}$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	10	020	016

Subtract with Carry**SBC**

Operation: $ACCX \leftarrow (ACCX) - (M) - (C)$

Description: Subtracts the contents of M and C from the contents of ACCX and places the result in ACCX.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
 C: Carry is set if the absolute value of the contents of memory plus previous carry is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	82	202	130
A DIR	3	2	92	222	146
A EXT	4	3	B2	262	178
A IND	5	2	A2	242	162
B IMM	2	2	C2	302	194
B DIR	3	2	D2	322	210
B EXT	4	3	F2	362	242
B IND	5	2	E2	342	226

SEC**Set Carry**

Operation: C bit \leftarrow 1

Description: Sets the carry bit in the processor condition codes register.

Condition Codes: H: Not affected.
I: Not affected.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Set.

Boolean Formulae for Condition Codes:

$$C = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0D	015	013

Set Interrupt Mask**SEI**Operation: I bit \leftarrow 1

Description: Sets the interrupt mask bit in the processor condition codes register. The microprocessor is inhibited from servicing an interrupt from a peripheral device, and will continue with execution of the instructions of the program, until the interrupt mask bit has been cleared.

Condition Codes: H: Not affected.
 I: Set.
 N: Not affected.
 Z: Not affected.
 V: Not affected.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$I = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0F	017	015

SEV**Set Two's Complement Overflow Bit**

Operation: $V \text{ bit} \leftarrow 1$

Description: Sets the two's complement overflow bit in the processor condition codes register.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Not affected.
 Z: Not affected.
 V: Set.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$V = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0B	013	011

Store Accumulator**STA**

Operation: $M \leftarrow (ACCX)$

Description: Stores the contents of ACCX in memory. The contents of ACCX remains unchanged.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the contents of ACCX is set; cleared otherwise.
 Z: Set if all bits of the contents of ACCX are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = X_7$$

$$Z = \overline{X}_7 \cdot \overline{X}_6 \cdot \overline{X}_5 \cdot \overline{X}_4 \cdot \overline{X}_3 \cdot \overline{X}_2 \cdot \overline{X}_1 \cdot \overline{X}_0$$

$$V = 0$$

Addressing Formats:

See Table A-2 (Page A-78).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A DIR	4	2	97	227	151
A EXT	5	3	B7	267	183
A IND	6	2	A7	247	167
B DIR	4	2	D7	327	215
B EXT	5	3	F7	367	247
B IND	6	2	E7	347	231

STS

Store Stack Pointer

Operation: $M \leftarrow (SPH)$
 $M + 1 \leftarrow (SPL)$

Description: Stores the more significant byte of the stack pointer in memory at the address specified by the program, and stores the less significant byte of the stack pointer at the next location in memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bit of the stack pointer is set; cleared otherwise.
 Z: Set if all bits of the stack pointer are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = SPH_7$$

$$Z = (\overline{SPH_7} \cdot \overline{SPH_6} \cdot \overline{SPH_5} \cdot \overline{SPH_4} \cdot \overline{SPH_3} \cdot \overline{SPH_2} \cdot \overline{SPH_1} \cdot \overline{SPH_0}) \cdot (\overline{SPL_7} \cdot \overline{SPL_6} \cdot \overline{SPL_5} \cdot \overline{SPL_4} \cdot \overline{SPL_3} \cdot \overline{SPL_2} \cdot \overline{SPL_1} \cdot \overline{SPL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-6 (Page A-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	9F	237	159
EXT	6	3	BF	277	191
IND	7	2	AF	257	175

Store Index Register**STX**

Operation: $M \leftarrow (IXH)$

$M + 1 \leftarrow (IXL)$

Description: Stores the more significant byte of the index register in memory at the address specified by the program, and stores the less significant byte of the index register at the next location in memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant bite of the index register is set; cleared otherwise.
 Z: Set if all bits of the index register are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = IXH_7$$

$$Z = (\overline{IXH_7} \cdot \overline{IXH_6} \cdot \overline{IXH_5} \cdot \overline{IXH_4} \cdot \overline{IXH_3} \cdot \overline{IXH_2} \cdot \overline{IXH_1} \cdot \overline{IXH_0}) \cdot (\overline{IXL_7} \cdot \overline{IXL_6} \cdot \overline{IXL_5} \cdot \overline{IXL_4} \cdot \overline{IXL_3} \cdot \overline{IXL_2} \cdot \overline{IXL_1} \cdot \overline{IXL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-6 (Page A-81).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	DF	337	223
EXT	6	3	FF	377	255
IND	7	2	EF	357	239

SUB

Subtract

Operation: $ACCX \leftarrow (ACCX) - (M)$

Description: Subtracts the contents of M from the contents of ACCX and places the result in ACCX.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if most significant bit of the result is set; cleared otherwise.
 Z: Set if all bits of the result are cleared; cleared otherwise.
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.
 C: Set if the absolute value of the contents of memory are larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 \cdot \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table A-1 (Page A-77).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	80	200	128
A DIR	3	2	90	220	144
A EXT	4	3	B0	260	176
A IND	5	2	A0	240	160
B IMM	2	2	C0	300	192
B DIR	3	2	D0	320	208
B EXT	4	3	F0	360	240
B IND	5	2	E0	340	224

Software Interrupt**SWI**

Operation:

$$PC \leftarrow (PC) + 0001$$

$$\downarrow (PCL), SP \leftarrow (SP) - 0001$$

$$\downarrow (PCH), SP \leftarrow (SP) - 0001$$

$$\downarrow (IXL), SP \leftarrow (SP) - 0001$$

$$\downarrow (IXH), SP \leftarrow (SP) - 0001$$

$$\downarrow (ACCA), SP \leftarrow (SP) - 0001$$

$$\downarrow (ACCB), SP \leftarrow (SP) - 0001$$

$$\downarrow (CC), SP \leftarrow (SP) - 0001$$

$$I \leftarrow 1$$

$$PCH \leftarrow (n-0005)$$

$$PCL \leftarrow (n-0004)$$

Description: The program counter is incremented (by 1). The program counter, index register, and accumulator A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

The interrupt mask bit is then set. The program counter is then loaded with the address stored in the software interrupt pointer at memory locations (n-5) and (n-4), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:

H: Not affected.

I: Set.

N: Not affected.

Z: Not affected.

V: Not affected.

C: Not affected.

Boolean Formula for Condition Codes:

$$I = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	12	1	3F	077	063

Software Interrupt**EXAMPLE****A. Before:**

CC = HINZVC (binary)

ACCB = 12 (Hex)

IXH = 56 (Hex)

ACCA = 34 (Hex)

IXL = 78 (Hex)

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
PC	→	\$5566	3F		SWI	
SP	→	\$EFFF				
		\$FFFA	D0			
		\$FFFB	55			

B. After:

PC → \$D055

SP → \$EFF8

\$EFF9 11HINZVC (binary)

\$EFFA 12

\$EFFB 34

\$EFFC 56

\$EFFD 78

\$EFFE 55

\$EFFF 67

Note: This example assumes that FFFF is the memory location addressed when all lines of the address bus go to the high state.

Transfer from Accumulator A to Accumulator B**TAB**Operation: $ACCB \leftarrow (ACCA)$

Description: Moves the contents of ACCA to ACCB. The former contents of ACCB are lost. The contents of ACCA are not affected.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the contents of the accumulator is set; cleared otherwise.
- Z: Set if all bits of the contents of the accumulator are cleared; cleared otherwise.
- V: Cleared.
- C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

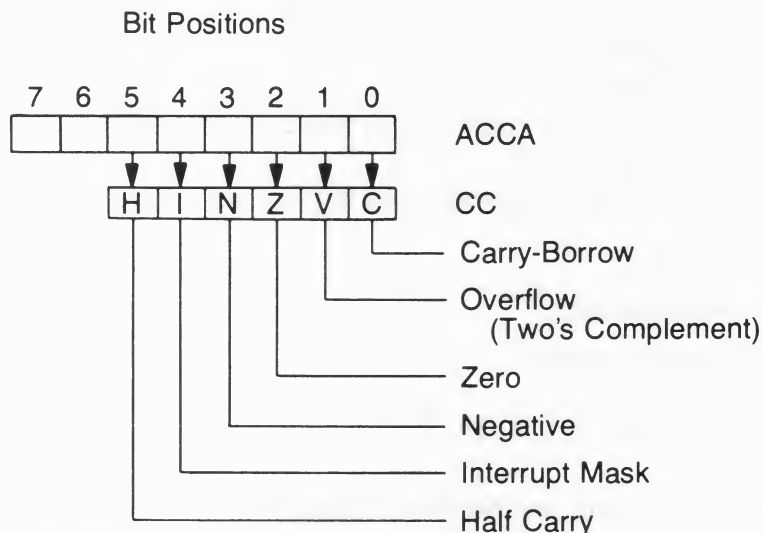
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	16	026	022

TAP

Transfer from Accumulator A to Processor Condition Codes Register

Operation: $CC \leftarrow (ACCA)$



Description: Transfers the contents of bit positions 0 thru 5 of accumulator A to the corresponding bit positions of the processor condition codes register. The contents of accumulator A remain unchanged.

Condition Codes: Set or reset according to the contents of the respective bits 0 thru 5 of accumulator A.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	06	006	006

Transfer from Accumulator B to Accumulator A**TBA**Operation: $ACCA \leftarrow (ACCB)$

Description: Moves the contents of ACCB to ACCA. The former contents of ACCA are lost. The contents of ACCB are not affected.

Condition Codes: H: Not affected.
 I: Not affected.
 N: Set if the most significant accumulator bit is set; cleared otherwise.
 Z: Set if all accumulator bits are cleared; cleared otherwise.
 V: Cleared.
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

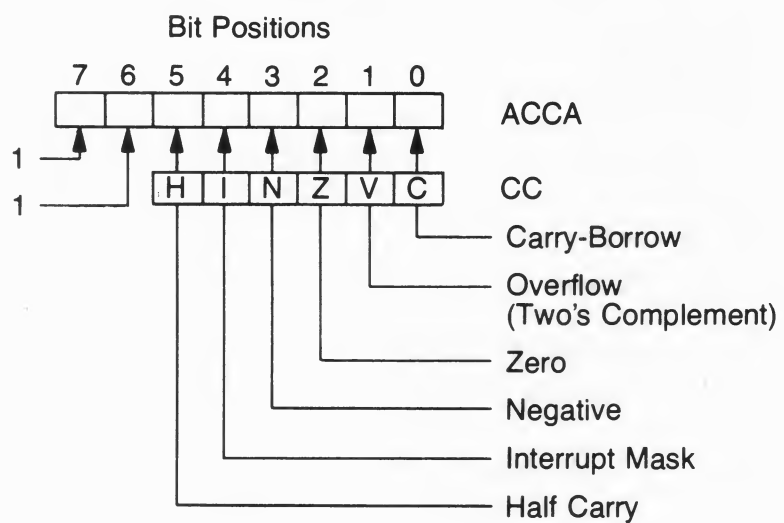
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	17	027	023

TPA

Transfer from Processor Condition Codes Register to Accumulator A

Operation: ACCA ← (CC)



Description: Transfers the contents of the processor condition codes register to corresponding bit positions 0 thru 5 of accumulator A. Bit positions 6 and 7 of accumulator A are set (i.e. go to the “1” state). The processor condition codes register remains unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	07	007	007

Test**TST**

Operation: (ACCX) – 00
(M) – 00

Description: Set condition codes N and Z according to the contents of ACCX or M.

Condition Codes: H: Not affected.
I: Not affected.
N: Set if most significant bit of the contents of ACCX or M is set; cleared otherwise.
Z: Set if all bits of the contents of ACCX or M are cleared; cleared otherwise.
V: Cleared.
C: Cleared.

Boolean Formulae for Condition Codes:

$$N = M_7$$

$$Z = \overline{M}_7 \cdot \overline{M}_6 \cdot \overline{M}_5 \cdot \overline{M}_4 \cdot \overline{M}_3 \cdot \overline{M}_2 \cdot \overline{M}_1 \cdot \overline{M}_0$$

$$V = 0$$

$$C = 0$$

Addressing Formats:

See Table A-3 (Page A-79).

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4D	115	077
B	2	1	5D	135	093
EXT	6	3	7D	175	125
IND	7	2	6D	155	109

TSX**Transfer from Stack Pointer to Index Register**

Operation: $IX \leftarrow (SP) + 0001$

Description: Loads the index register with one plus the contents of the stack pointer. The contents of the stack pointer remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	30	060	048

Transfer From Index Register to Stack Pointer**TXS**Operation: $SP \leftarrow (IX) - 0001$ Description: Loads the stack pointer with the contents of the index register, minus one.
The contents of the index register remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	35	.065	053

WAI

Wait for Interrupt

Operation: $PC \leftarrow (PC) + 0001$
 $\downarrow (PCL), SP \leftarrow (SP) - 0001$
 $\downarrow (PCH), SP \leftarrow (SP) - 0001$
 $\downarrow (IXL), SP \leftarrow (SP) - 0001$
 $\downarrow (IXH), SP \leftarrow (SP) - 0001$
 $\downarrow (ACCA), SP \leftarrow (SP) - 0001$
 $\downarrow (ACCB), SP \leftarrow (SP) - 0001$
 $\downarrow (CC), SP \leftarrow (SP) - 0001$

Condition Codes: Not affected.

Description: The program counter is incremented (by 1). The program counter, index register, and accumulators A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

Execution of the program is then suspended until an interrupt from a peripheral device is signalled, by the interrupt request control input going to a low state.

When an interrupt is signalled on the interrupt request line, and provided the I bit is clear, execution proceeds as follows. The interrupt mask bit is set. The program counter is then loaded with the address stored in the internal interrupt pointer at memory locations (n-7) and (n-6), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes: H: Not affected.
I: Not affected until an interrupt request signal is detected on the interrupt request control line. When the interrupt request is received the I bit is set and further execution takes place, provided the I bit was initially clear.
N: Not affected.
Z: Not affected.
V: Not affected.
C: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	9	1	3E	076	062

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
IMMediate	CCC A #number CCC A #symbol CCC A #expression CCC A #'C	CCC B #number CCC B #symbol CCC B #expression CCC B #'C
DIRect or EXTended	CCC A number CCC A symbol CCC A expression	CCC B number CCC B symbol CCC B expression
INDexed	CCC A X CCC Z ,X CCC A number,X CCC A symbol,X CCC A expression,X	CCC B X CCC B ,X CCC B number,X CCC B symbol,X CCC B expression,X

- Notes: 1. CCC = mnemonic operator of source instruction.
 2. "symbol" may be the special symbol "*".
 3. "expression" may contain the special symbol "*".
 4. space may be omitted before A or B.

Applicable to the following source instructions:

ADC ADD AND BIT CMP
 EOR LDA ORA SBC SUB

*Special symbol indicating program-counter.

TABLE A-1. Addressing Formats (1)

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
DIRect or EXTended	STA A number STA A symbol STA A expression	STA B number STA B symbol STA B expression
INDexed	STA A X STA A ,X STA A number,X STA A symbol,X STA A expression,X	STA B X STA B ,X STA B number,X STA B symbol,X STA B expression,X

Notes: 1. "symbol" may be the special symbol "**".
 2. "expression" may contain the special symbol "**".
 3. Space may be omitted before A or B.

Applicable to the source instruction:

STA

*Special symbol indicating program-counter.

TABLE A-2. Addressing Formats (2)

Operand or Addressing Mode	Formats
Accumulator A	CCC A
Accumulator B	CCC B
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
 2. "symbol" may be the special symbol "*".
 3. "expression" may contain the special symbol "*".
 4. Space may be omitted before A or B.

Applicable to the following source instructions:

ASL ASR CLR COM DEC INC
 LSR NEG ROL ROR TST

*Special symbol indicating program-counter.

TABLE A-3. Addressing Formats (3)

Operand	Formats
Accumulator A	CCC A
Accumulator B	CCC B

Notes: 1. CCC = mnemonic operator of source instruction.
 2. Space may be omitted before A or B.

Applicable to the following source instructions:

PSH PUL

TABLE A-4. Addressing Formats (4)

Addressing Mode	Formats
IMMediate	CCC #number CCC #symbol CCC #expression CCC #'C
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
 2. "symbol" may be the special symbol "*".
 3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:

CPX LDS LDX

*Special symbol indicating program-counter.

TABLE A-5. Addressing Formats (5)

Addressing Mode	Formats
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
2. "symbol" may be the special symbol "*".
3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:

STS STX

*Special symbol indicating program-counter.

TABLE A-6. Addressing Formats (6)

Addressing Mode	Formats
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

Notes: 1. CCC = mnemonic operator of source instruction.
 2. "symbol" may be the special symbol "*".
 3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:

JMP JSR

*Special symbol indicating program-counter.

TABLE A-7. Addressing Formats (7)

Addressing Mode	Formats
RELative	CCC number CCC symbol CCC expression

Notes: 1. CCC = mnemonic operator of source instruction.
 2. "symbol" may be the special symbol "*".
 3. "expression" may contain the special symbol "*".

Applicable to the following source instructions:

BCC BCS BEQ BGE BGT BHI BLE BLS
 BLT BMI BNE BPL BRA BSR BVC BVS

*Special symbol indicating program-counter.

TABLE A-8. Addressing Formats (8)

Appendix B

DATA SHEETS

CONTENTS

MC6800 Data Sheet	Page B-3
MC6801 Data Sheet	Page B-21
MC6802 Data Sheet	Page B-53
MC6803 Data Sheet	Page B-73
MC6805 Data Sheet	Page B-103
MC6808 Data Sheet	Page B-127
MC6809 Data Sheet	Page B-147
Positive Powers of 2	Page B-171
Negative Powers of 2	Page B-174
Positive Powers of 8	Page B-175
Positive Powers of 16	Page B-175
Negative Powers of 16	Page B-175


MOTOROLA
Semiconductors

BOX 20912 • PHOENIX, ARIZONA 85036

MICROPROCESSING UNIT (MPU)

The MC6800 is a monolithic 8-bit microprocessor forming the central control function for Motorola's M6800 family. Compatible with TTL, the MC6800, as with all M6800 system parts, requires only one +5.0-volt power supply, and no external TTL devices for bus interface.

The MC6800 is capable of addressing 65K bytes of memory with its 16-bit address lines. The 8-bit data bus is bidirectional as well as 3-state, making direct memory addressing and multiprocessing applications realizable.

- Eight-Bit Parallel Processing
- Bi-Directional Data Bus
- Sixteen-Bit Address Bus — 65K Bytes of Addressing
- 72 Instructions — Variable Length
- Seven Addressing Modes — Direct, Relative, Immediate, Indexed, Extended, Implied and Accumulator
- Variable Length Stack
- Vectored Restart
- Maskable Interrupt Vector
- Separate Non-Maskable Interrupt — Internal Registers Saved In Stack
- Six Internal Registers — Two Accumulators, Index Register, Program Counter, Stack Pointer and Condition Code Register
- Direct Memory Addressing (DMA) and Multiple Processor Capability
- Clock Rates as High as 1 MHz
- Simple Bus Interface Without TTL
- Halt and Single Instruction Execution Capability

MC6800

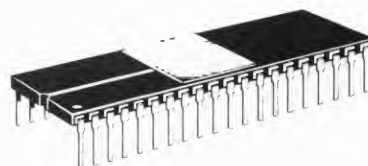
(0 to 70°C; L or P Suffix)

MC6800C

(-40 to 85°C; L Suffix only)

MOS

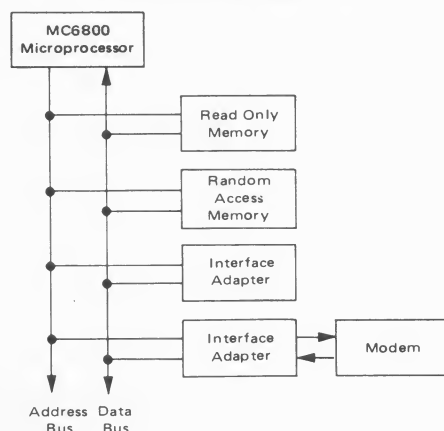
(N-CHANNEL, SILICON-GATE)

MICROPROCESSOR


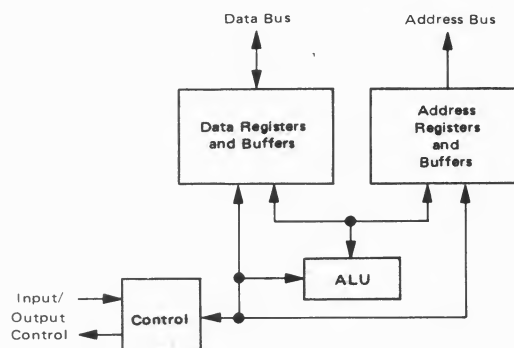
L SUFFIX
 CERAMIC PACKAGE
 CASE 715

NOT SHOWN: **P SUFFIX**
 PLASTIC PACKAGE
 CASE 711

**M6800 MICROCOMPUTER FAMILY
 BLOCK DIAGRAM**



**MC6800 MICROPROCESSOR
 BLOCK DIAGRAM**



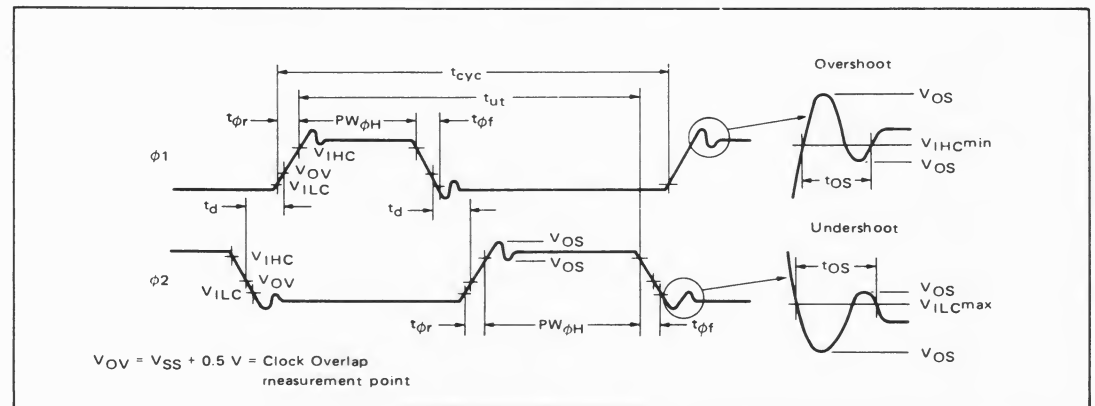
MC6800

ELECTRICAL CHARACTERISTICS ($V_{CC} = 5.0 \text{ V} \pm 5\%$, $V_{SS} = 0$, $T_A = 0$ to 70°C unless otherwise noted.)

Characteristic		Symbol	Min	Typ	Max	Unit
Input High Voltage	Logic $\phi 1, \phi 2$	V_{IH} V_{IHC}	$V_{SS} + 2.0$ $V_{CC} - 0.3$	— —	V_{CC} $V_{CC} + 0.1$	Vdc
Input Low Voltage	Logic $\phi 1, \phi 2$	V_{IL} V_{ILC}	$V_{SS} - 0.3$ $V_{SS} - 0.1$	— —	$V_{SS} + 0.8$ $V_{SS} + 0.3$	Vdc
Clock Overshoot/Undershoot — Input High Level — Input Low Level		V_{OS}	$V_{CC} - 0.5$ $V_{SS} - 0.5$	— —	$V_{CC} + 0.5$ $V_{SS} + 0.5$	Vdc
Input Leakage Current ($V_{in} = 0$ to 5.25 V , $V_{CC} = \text{max}$) ($V_{in} = 0$ to 5.25 V , $V_{CC} = 0.0 \text{ V}$)	Logic* $\phi 1, \phi 2$	I_{in}	— —	1.0 —	2.5 100	μAdc
Three-State (Off State) Input Current ($V_{in} = 0.4$ to 2.4 V , $V_{CC} = \text{max}$)	D0-D7 A0-A15,R/W	I_{TSI}	— —	2.0 —	10 100	μAdc
Output High Voltage ($I_{Load} = -205 \mu\text{Adc}$, $V_{CC} = \text{min}$) ($I_{Load} = -145 \mu\text{Adc}$, $V_{CC} = \text{min}$) ($I_{Load} = -100 \mu\text{Adc}$, $V_{CC} = \text{min}$)	D0-D7 A0-A15,R/W,VMA BA	V_{OH}	$V_{SS} + 2.4$ $V_{SS} + 2.4$ $V_{SS} + 2.4$	— — —	— — —	Vdc
Output Low Voltage ($I_{Load} = 1.6 \text{ mAdc}$, $V_{CC} = \text{min}$)		V_{OL}	—	—	$V_{SS} + 0.4$	Vdc
Power Dissipation		P_D	—	0.600	1.2	W
Capacitance [#] ($V_{in} = 0$, $T_A = 25^\circ\text{C}$, $f = 1.0 \text{ MHz}$)	$\phi 1, \phi 2$ TSC DBE D0-D7 Logic Inputs A0-A15,R/W,VMA	C_{in} C_{out}	80 — — — — —	120 — 7.0 10 6.5 —	160 15 10 12.5 8.5 12	pF pF
Frequency of Operation		f	0.1	—	1.0	MHz
Clock Timing (Figure 1)						
Cycle Time		t_{cyc}	1.0	—	10	μs
Clock Pulse Width (Measured at $V_{CC} - 0.3 \text{ V}$)	$\phi 1$ $\phi 2$	$PW_{\phi H}$	430 450	— —	4500 4500	ns
Total $\phi 1$ and $\phi 2$ Up Time		t_{ut}	940	—	—	ns
Rise and Fall Times (Measured between $V_{SS} + 0.3 \text{ V}$ and $V_{CC} - 0.3 \text{ V}$)	$\phi 1, \phi 2$	$t_{\phi r}, t_{\phi f}$	5.0	—	50	ns
Delay Time or Clock Separation (Measured at $V_{OV} = V_{SS} + 0.5 \text{ V}$)		t_d	0	—	9100	ns
Overshoot Duration		t_{OS}	0	—	40	ns

*Except \overline{TRQ} and \overline{NMI} , which require $3 \text{ k}\Omega$ pullup load resistors for wire-OR capability at optimum operation.[#]Capacitances are periodically sampled rather than 100% tested.

FIGURE 1 — CLOCK TIMING WAVEFORM



MC6800
MAXIMUM RATINGS

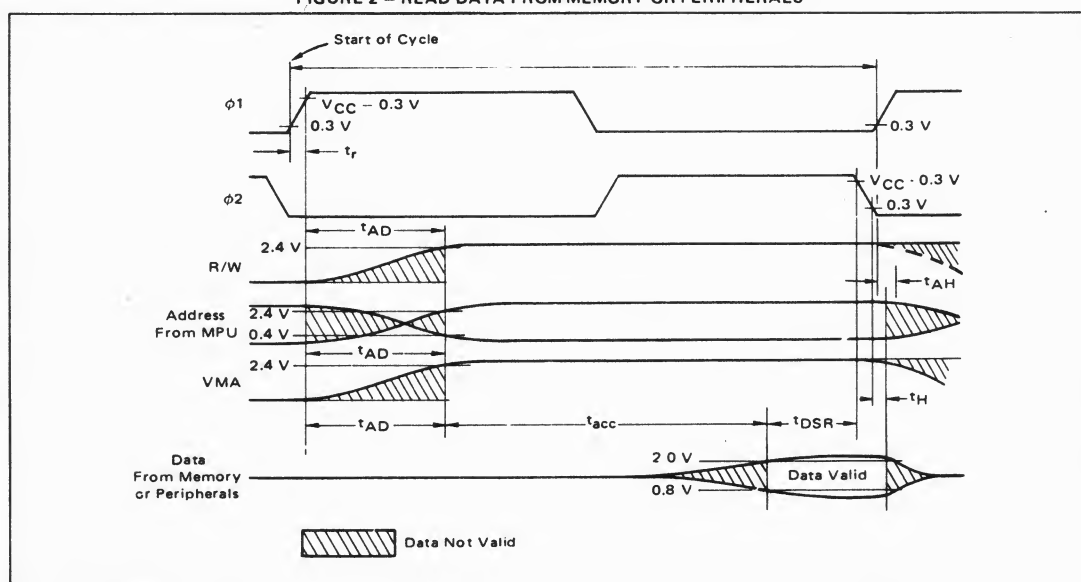
Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	Vdc
Input Voltage	V_{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C
Thermal Resistance	θ_{JA}	70	°C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

READ/WRITE TIMING Figures 2 and 3, $f = 1.0$ MHz, Load Circuit of Figure 6.

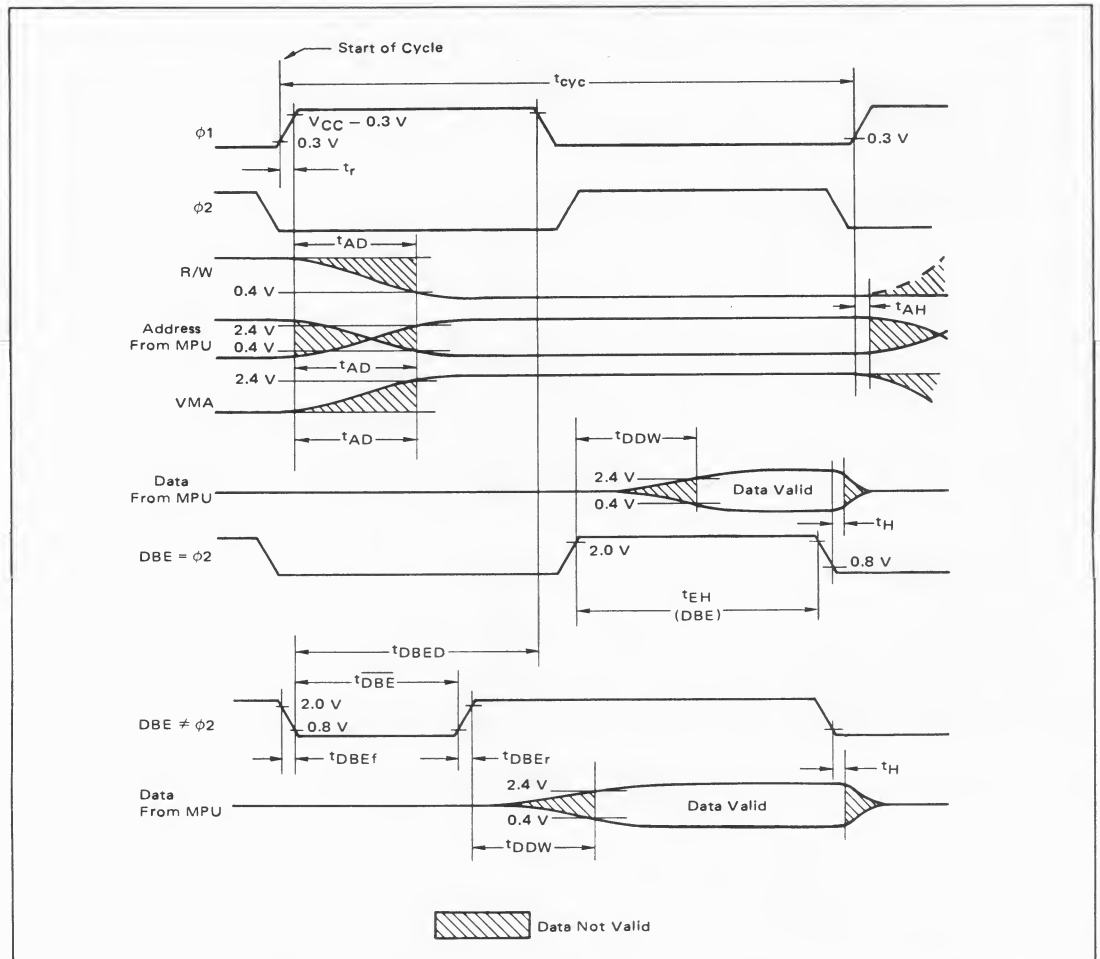
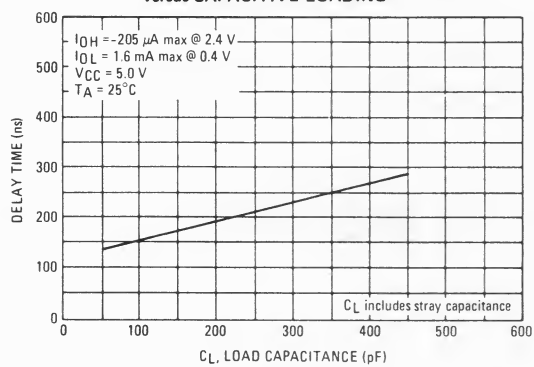
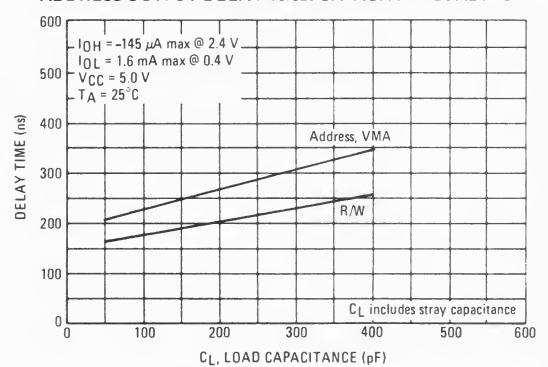
Characteristic	Symbol	Min	Typ	Max	Unit
Address Delay	t_{AD}	—	220	300	ns
Peripheral Read Access Time $t_{acc} = t_{ut} - (t_{AD} + t_{DSR})$	t_{acc}	—	—	540	ns
Data Setup Time (Read)	t_{DSR}	100	—	—	ns
Input Data Hold Time	t_H	10	—	—	ns
Output Data Hold Time	t_H	10	25	—	ns
Address Hold Time (Address, R/W, VMA)	t_{AH}	50	75	—	ns
Enable High Time for DBE Input	t_{EH}	450	—	—	ns
Data Delay Time (Write)	t_{DDW}	—	165	225	ns
Processor Controls*					
Processor Control Setup Time	t_{PCS}	200	—	—	ns
Processor Control Rise and Fall Time	t_{PCr}, t_{PCf}	—	—	100	ns
Bus Available Delay	t_{BA}	—	—	300	ns
Three State Enable	t_{TSE}	—	—	40	ns
Three State Delay	t_{TSD}	—	—	700	ns
Data Bus Enable Down Time During ϕ_1 Up Time (Figure 3)	t_{DBE}	150	—	—	ns
Data Bus Enable Delay (Figure 3)	t_{DBED}	300	—	—	ns
Data Bus Enable Rise and Fall Times (Figure 3)	t_{DBEr}, t_{DBEf}	—	—	25	ns

*Additional information is given in Figures 12 through 16 of the Family Characteristics — see pages 17 through 20.

FIGURE 2 — READ DATA FROM MEMORY OR PERIPHERALS

MOTOROLA Semiconductor Products Inc.

MC6800

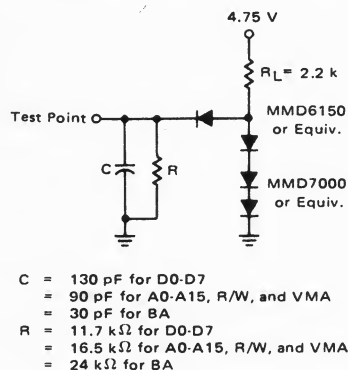
FIGURE 3 — WRITE IN MEMORY OR PERIPHERALS

FIGURE 4 — TYPICAL DATA BUS OUTPUT DELAY
versus CAPACITIVE LOADINGFIGURE 5 — TYPICAL READ/WRITE, VMA, AND
ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING

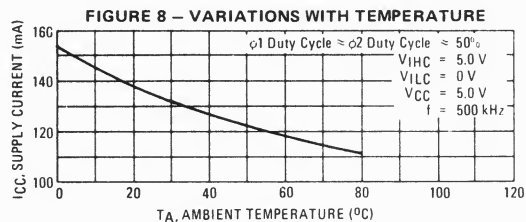
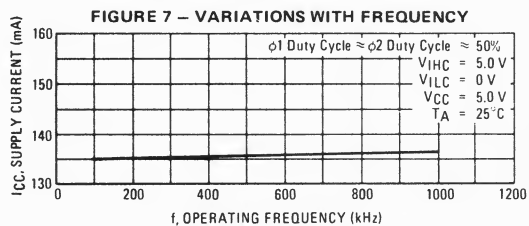
MOTOROLA Semiconductor Products Inc.

MC6800

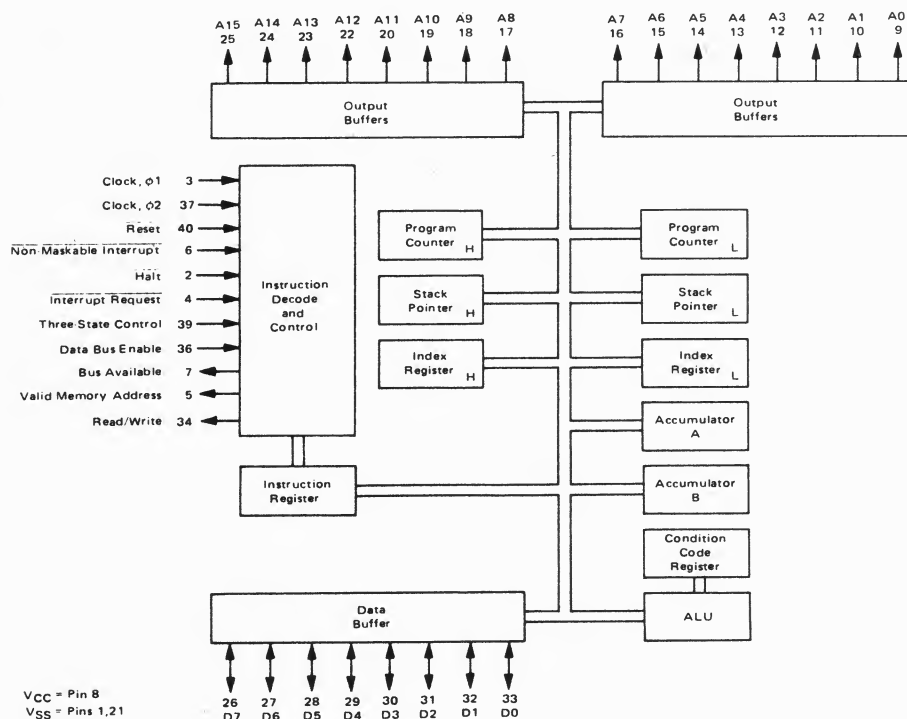
FIGURE 6 – BUS TIMING TEST LOAD



TYPICAL POWER SUPPLY CURRENT



EXPANDED BLOCK DIAGRAM



MOTOROLA Semiconductor Products Inc.

MC6800

MPU SIGNAL DESCRIPTION

Proper operation of the MPU requires that certain control and timing signals be provided to accomplish specific functions and that other signal lines be monitored to determine the state of the processor.

Clocks Phase One and Phase Two ($\phi 1, \phi 2$) — Two pins are used for a two-phase non-overlapping clock that runs at the V_{CC} voltage level.

Address Bus (A0-A15) — Sixteen pins are used for the address bus. The outputs are three-state bus drivers capable of driving one standard TTL load and 130 pF. When the output is turned off, it is essentially an open circuit. This permits the MPU to be used in DMA applications.

Data Bus (D0-D7) — Eight pins are used for the data bus. It is bi-directional, transferring data to and from the memory and peripheral devices. It also has three-state output buffers capable of driving one standard TTL load and 130 pF.

Halt — When this input is in the low state, all activity in the machine will be halted. This input is level sensitive. In the halt mode, the machine will stop at the end of an instruction, Bus Available will be at a one level, Valid Memory Address will be at a zero, and all other three-state lines will be in the three-state mode.

Transition of the $\overline{\text{Halt}}$ line must not occur during the last 250 ns of phase one. To insure single instruction operation, the $\overline{\text{Halt}}$ line must go high for one Clock cycle.

Three-State Control (TSC) — This input causes all of the address lines and the Read/Write line to go into the off or high impedance state. This state will occur 700 ns after $TSC = 2.0$ V. The Valid Memory Address and Bus Available signals will be forced low. The data bus is not affected by TSC and has its own enable (Data Bus Enable). In DMA applications, the Three-State Control line should be brought high on the leading edge of the Phase One Clock. The $\phi 1$ clock must be held in the high state and the $\phi 2$ in the low state for this function to operate properly. The address bus will then be available for other devices to directly address memory. Since the MPU is a dynamic device, it can be held in this state for only 4.5 μ s or destruction of data will occur in the MPU.

Read/Write (R/W) — This TTL compatible output signals the peripherals and memory devices whether the MPU is in a Read (high) or Write (low) state. The normal standby state of this signal is Read (high). Three-State Control going high will turn Read/Write to the off (high impedance) state. Also, when the processor is halted, it will be in the off state. This output is capable of driving one standard TTL load and 90 pF.

Valid Memory Address (VMA) — This output indicates to peripheral devices that there is a valid address on the address bus. In normal operation, this signal should be utilized for enabling peripheral interfaces such as the PIA and ACIA. This signal is not three-state. One standard TTL load and 90 pF may be directly driven by this active high signal.

Data Bus Enable (DBE) — This input is the three-state control signal for the MPU data bus and will enable the bus drivers when in the high state. This input is TTL compatible; however in normal operation, it would be driven by the phase two clock. During an MPU read cycle, the data bus drivers will be disabled internally. When it is desired that another device control the data bus such as in Direct Memory Access (DMA) applications, DBE should be held low.

Bus Available (BA) — The Bus Available signal will normally be in the low state; when activated, it will go to the high state indicating that the microprocessor has stopped and that the address bus is available. This will occur if the $\overline{\text{Halt}}$ line is in the low state or the processor is in the WAIT state as a result of the execution of a WAIT instruction. At such time, all three-state output drivers will go to their off state and other outputs to their normally inactive level. The processor is removed from the WAIT state by the occurrence of a maskable (mask bit $I = 0$) or nonmaskable interrupt. This output is capable of driving one standard TTL load and 30 pF.

Interrupt Request ($\overline{\text{IRQ}}$) — This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further interrupts may occur. At the end of the cycle, a 16-bit address will be loaded that points to a vectored address which is located in memory locations FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine in memory.

The $\overline{\text{Halt}}$ line must be in the high state for interrupts to be serviced. Interrupts will be latched internally while $\overline{\text{Halt}}$ is low.

The $\overline{\text{IRQ}}$ has a high impedance pullup device internal to the chip; however a 3 k Ω external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

Reset — This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. If a high level is detected on the input, this will signal the MPU to begin the restart sequence. This will start execution of a routine to initialize the processor from its reset condition. All the higher order address lines will be forced high. For the restart, the last two (FFFE, FFFF) locations in memory will be used to load the program that is addressed by the program counter. During the restart routine, the interrupt mask bit is set and must be reset before the MPU can be interrupted by $\overline{\text{IRQ}}$.


MOTOROLA Semiconductor Products Inc.

MC6800

Figure 9 shows the initialization of the microprocessor after restart. Reset must be held low for at least eight clock periods after V_{CC} reaches 4.75 volts. If $\overline{\text{Reset}}$ goes high prior to the leading edge of ϕ_2 , on the next ϕ_1 the first restart memory vector address (FFFE) will appear on the address lines. This location should contain the higher order eight bits to be stored into the program counter. Following, the next address FFFF should contain the lower order eight bits to be stored into the program counter.

Non-Maskable Interrupt (NMI) — A low-going edge on this input requests that a non-mask-interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the $\overline{\text{NMI}}$ signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. At the end of the cycle, a 16-bit address will be loaded that points to a vectored address which is located in memory locations FFFC and FFFD. An address loaded at these locations causes the MPU to branch to a non-maskable interrupt routine in memory.

NMI has a high impedance pullup resistor internal to the chip; however a $3\text{ k}\Omega$ external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

Inputs $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ are hardware interrupt lines that are sampled during ϕ_2 and will start the interrupt routine on the ϕ_1 following the completion of an instruction.

Figure 10 is a flow chart describing the major decision paths and interrupt vectors of the microprocessor. Table 1 gives the memory map for interrupt vectors.

FIGURE 9 — INITIALIZATION OF MPU AFTER RESTART

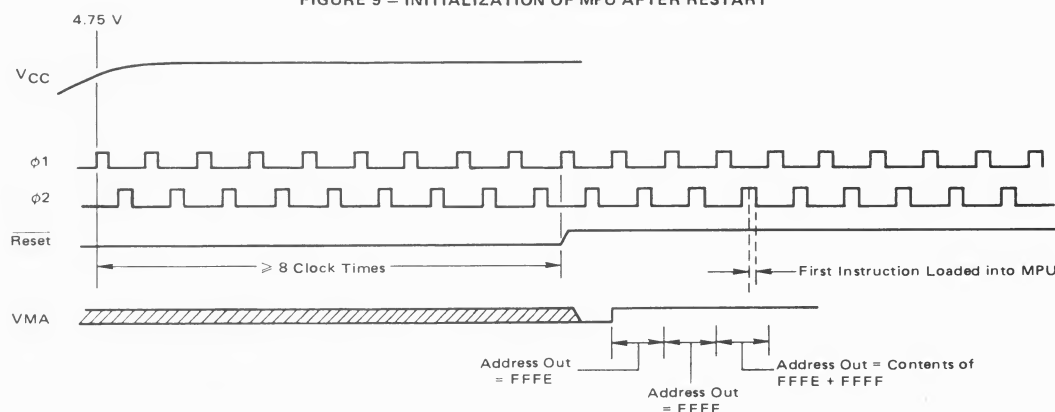


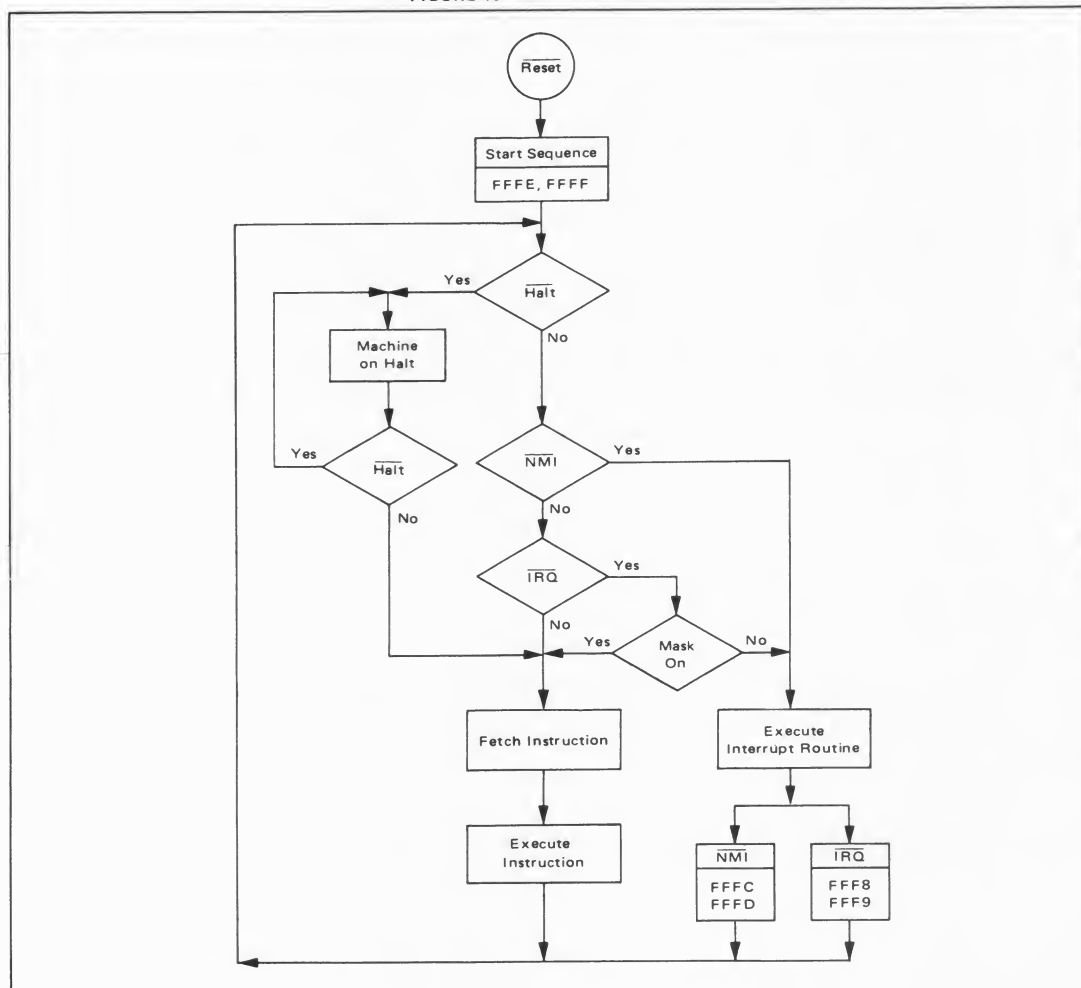
TABLE 1 — MEMORY MAP FOR INTERRUPT VECTORS

Vector		Description
MS	LS	
FFFE	FFFF	Restart
FFFC	FFFD	Non-maskable Interrupt
FFFA	FFFB	Software Interrupt
FFF8	FFF9	Interrupt Request



MC6800

FIGURE 10 — MPU FLOW CHART



MPU REGISTERS

The MPU has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 11).

Program Counter — The program counter is a two byte (16-bits) register that points to the current program address.

Stack Pointer — The stack pointer is a two byte register that contains the address of the next available location in an external push-down/pop-up stack. This stack is normally a random access Read/Write memory that may

have any location (address) that is convenient. In those applications that require storage of information in the stack when power is lost, the stack must be non-volatile.

Index Register — The index register is a two byte register that is used to store data or a sixteen bit memory address for the Indexed mode of memory addressing.

Accumulators — The MPU contains two 8-bit accumulators that are used to hold operands and results from an arithmetic logic unit (ALU).



MOTOROLA Semiconductor Products Inc.

MC6800

FIGURE 11 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT

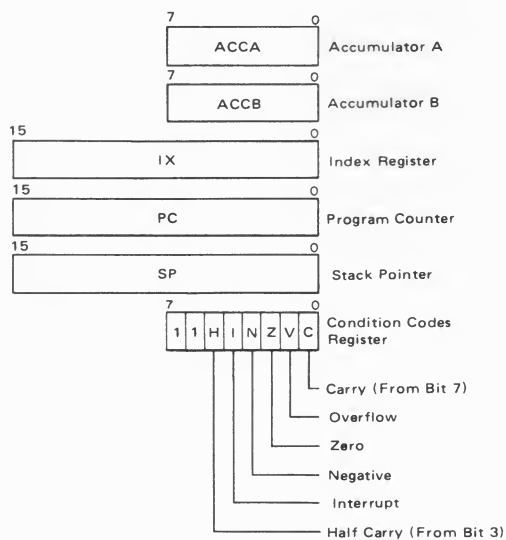
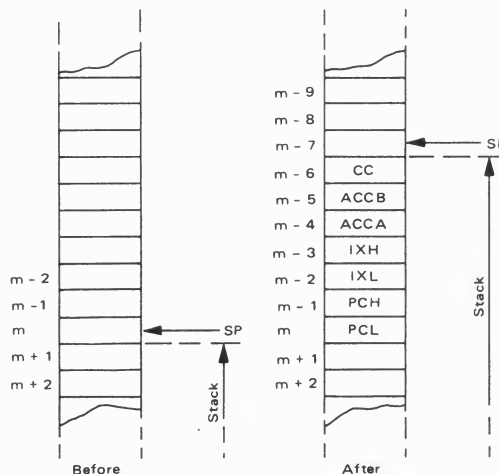


FIGURE 12 — SAVING THE STATUS OF THE MICROPROCESSOR IN THE STACK

SP = Stack Pointer
CC = Condition Codes (Also called the Processor Status Byte)
ACCB = Accumulator B
ACCA = Accumulator A
IXH = Index Register, Higher Order 8 Bits
IXL = Index Register, Lower Order 8 Bits
PCH = Program Counter, Higher Order 8 Bits
PCL = Program Counter, Lower Order 8 Bits



MOTOROLA Semiconductor Products Inc.

MC6800

Condition Code Register — The condition code register indicates the results of an Arithmetic Logic Unit operation: Negative (N), Zero (Z), Overflow (V), Carry from bit 7 (C), and half carry from bit 3 (H). These bits of the Condition Code Register are used as testable conditions for the conditional branch instructions. Bit 4 is the interrupt mask bit (I). The unused bits of the Condition Code Register (b6 and b7) are ones.

Figure 12 shows the order of saving the microprocessor status within the stack.

MPU INSTRUCTION SET

The MC6800 has a set of 72 different instructions. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, conditional or unconditional branch, interrupt and stack manipulation instructions (Tables 2 thru 6).

MPU ADDRESSING MODES

The MC6800 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 7 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 1 MHz, these times would be microseconds.

Accumulator (ACCX) Addressing — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

Immediate Addressing — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MPU addresses

this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

Direct Addressing — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

Extended Addressing — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

Indexed Addressing — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

Implied Addressing — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

Relative Addressing — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.

TABLE 2 — MICROPROCESSOR INSTRUCTION SET — ALPHABETIC SEQUENCE

ABA	Add Accumulators	CLR	Clear	PUL	Pull Data
ADC	Add with Carry	CLV	Clear Overflow	ROL	Rotate Left
ADD	Add	CMP	Compare	ROR	Rotate Right
AND	Logical And	COM	Complement	RTI	Return from Interrupt
ASL	Arithmetic Shift Left	CPX	Compare Index Register	RTS	Return from Subroutine
ASR	Arithmetic Shift Right	DAA	Decimal Adjust	SBA	Subtract Accumulators
BCC	Branch if Carry Clear	DEC	Decrement	SBC	Subtract with Carry
BCS	Branch if Carry Set	DES	Decrement Stack Pointer	SEC	Set Carry
BEQ	Branch if Equal to Zero	DEX	Decrement Index Register	SEI	Set Interrupt Mask
BGE	Branch if Greater or Equal Zero	EOR	Exclusive OR	SEV	Set Overflow
BGT	Branch if Greater than Zero	INC	Increment	STA	Store Accumulator
BHI	Branch if Higher	INS	Increment Stack Pointer	STS	Store Stack Register
BIT	Bit Test	INX	Increment Index Register	STX	Store Index Register
BLE	Branch if Less or Equal	JMP	Jump	SUB	Subtract
BLS	Branch if Lower or Same	JSR	Jump to Subroutine	SWI	Software Interrupt
BLT	Branch if Less than Zero	LDA	Load Accumulator	TAB	Transfer Accumulators
BMI	Branch if Minus	LDS	Load Stack Pointer	TAP	Transfer Accumulators to Condition Code Reg.
BNE	Branch if Not Equal to Zero	LDX	Load Index Register	TBA	Transfer Accumulators
BPL	Branch if Plus	LSR	Logical Shift Right	TPA	Transfer Condition Code Reg. to Accumulator
BRA	Branch Always	NEG	Negate	TST	Test
BSR	Branch to Subroutine	NOP	No Operation	TSX	Transfer Stack Pointer to Index Register
BVC	Branch if Overflow Clear	ORA	Inclusive OR Accumulator	TXS	Transfer Index Register to Stack Pointer
BVS	Branch if Overflow Set	PSH	Push Data	WAI	Wait for Interrupt
CBA	Compare Accumulators				
CLC	Clear Carry				
CLI	Clear Interrupt Mask				


MOTOROLA Semiconductor Products Inc.

MC6800

TABLE 3 — ACCUMULATOR AND MEMORY INSTRUCTIONS

OPERATIONS		MNEMONIC	ADDRESSING MODES								BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)		COND. CODE REG.						
			IMMED		DIRECT		INDEX		EXTND				IMPLIED		5	4	3	2	1
			OP	~	OP	~	OP	~	OP	~	OP	~	OP	~	H	I	N	Z	V
Add	ADDA	98 2 2		98 3 2		A8 5 2		B8 4 3				A + M · A							
	ADDB	C8 2 2		D8 3 2		E8 5 2		F8 4 3				B + M · B							
Add Acmltrs	ABA									18 2 1		A + B · A							
Add with Carry	ADCA	89 2 2		99 3 2		A9 5 2		B9 4 3				A + M + C · A							
	ADCB	C9 2 2		D9 3 2		E9 5 2		F9 4 3				B + M + C · B							
And	ANDA	84 2 2		94 3 2		A4 5 2		B4 4 3				A · M · A							
	ANDB	C4 2 2		D4 3 2		E4 5 2		F4 4 3				B · M · B							
Bit Test	BITA	85 2 2		95 3 2		A5 5 2		B5 4 3				A · M							
	BITB	C5 2 2		D5 3 2		E5 5 2		F5 4 3				B · M							
Clear	CLR					6F 7 2		7F 6 3				00 · M							
	CLRA									4F 2 1		00 · A							
	CLRB									5F 2 1		00 · B							
Compare	CMPA	81 2 2		91 3 2		A1 5 2		B1 4 3				A · M							
	CMPB	C1 2 2		D1 3 2		E1 5 2		F1 4 3				B · M							
Compare Acmltrs	CBA									11 2 1		A · B							
Complement, 1's	CUM					63 7 2		73 6 3				M · M							
	CDMA									43 2 1		Ā · A							
	CDMB									53 2 1		B · B							
Complement, 2's (Negate)	NEG					60 7 2		70 6 3				00 M · M							
	NEGA									40 2 1		00 A · A							
	NEGB									50 2 1		00 B · B							
Decimal Adjust, A	DAA									19 2 1		Converts Binary Add. of BCD Characters into BCD Format							
Decrement	DEC					6A 7 2		7A 6 3				M - 1 · M							
	DECA									4A 2 1		A - 1 · A							
	DECB									5A 2 1		B - 1 · B							
Exclusive OR	EDRA	88 2 2		98 3 2		A8 5 2		B8 4 3				A ⊕ M · A							
	EDRB	C8 2 2		D8 3 2		E8 5 2		F8 4 3				B ⊕ M · B							
Increment	INC					6C 7 2		7C 6 3				M + 1 · M							
	INCA									4C 2 1		A + 1 · A							
	INCB									5C 2 1		B + 1 · B							
Load Acmltr	LDAA	86 2 2		96 3 2		A6 5 2		B6 4 3				M · A							
	LDAB	C6 2 2		D6 3 2		E6 5 2		F6 4 3				M · B							
Or, Inclusive	ORAA	8A 2 2		9A 3 2		AA 5 2		BA 4 3				A + M · A							
	ORAB	CA 2 2		DA 3 2		EA 5 2		FA 4 3				B + M · B							
Push Data	PSHA									36 4 1		A · M _{SP} , SP - 1 · SP							
	PSHB									37 4 1		B · M _{SP} , SP - 1 · SP							
Pull Data	PULA									32 4 1		SP + 1 · SP, M _{SP} · A							
	PULB									33 4 1		SP + 1 · SP, M _{SP} · B							
Rotate Left	RDL					69 7 2		79 6 3				M							
	ROLA									49 2 1		A							
	ROLB									59 2 1		B							
Rotate Right	RDR					66 7 2		76 6 3				M							
	RORA									46 2 1		A							
	RORB									56 2 1		B							
Shift Left, Arithmetic	ASL					68 7 2		78 6 3				M							
	ASLA									48 2 1		A							
	ASLB									58 2 1		B							
Shift Right, Arithmetic	ASR					67 7 2		77 6 3				M							
	ASRA									47 2 1		A							
	ASRB									57 2 1		B							
Shift Right, Logic	LSR					64 7 2		74 6 3				M							
	LSRA									44 2 1		A							
	LSRB									54 2 1		B							
Store Acmltr	STAA			97 4 2		A/ 6 2		B/ 5 3				A · M							
	STAB			D7 4 2		E/ 6 2		F/ 5 3				B · M							
Subtract	SUBA	80 2 2		90 3 2		A0 5 2		B0 4 3				A - M · A							
	SUBB	C0 2 2		D0 3 2		E0 5 2		F0 4 3				B - M · B							
Subtract Acmltrs	SBA									10 2 1		A - B · A							
Subtr. with Carry	SBCA	82 2 2		92 3 2		A2 5 2		B2 4 3				A - M · C · A							
	SBCB	C2 2 2		D2 3 2		E2 5 2		F2 4 3				B - M · C · B							
Transfer Acmltrs	TAB									16 2 1		A · B							
	TBA									17 2 1		B · A							
Test, Zero or Minus	TST					6D 7 2		7D 6 3				M 00							
	TSTA									40 2 1		A 00							
	TSTB									50 2 1		B 00							

LEGEND:

OP Operation Code (Hexadecimal),
~ Number of MPU Cycles,
= Number of Program Bytes;
+ Arithmetic Plus;
- Arithmetic Minus;
• Boolean AND;

Msp Contents of memory location pointed to by Stack Pointer;

+ Boolean Inclusive OR;
⊕ Boolean Exclusive OR;
M Complement of M;
• Transfer Into;
0 Bit Zero;
00 Byte Zero;

Note Accumulator addressing mode instructions are included in the column for IMPLIED addressing

CONDITION CODE SYMBOLS:

H Half carry from bit 3;
I Interrupt mask;
N Negative (sign bit);
Z Zero (byte);
V Overflow, 2's complement;
C Carry from bit 7;
R Reset Always;
S Set Always;
• Test and set if true, cleared otherwise;
• Not Affected



MOTOROLA Semiconductor Products Inc.

MC6800

TABLE 4 — INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

																		COND. CODE REG						
POINTER OPERATIONS		MNEMONIC	IMMED			DIRECT			INDEX			EXTND			IMPLIED			BOOLEAN/ARITHMETIC OPERATION	5	4	3	2	1	0
			OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=		H	I	N	Z	V	O
Compare Index Reg	CPX	8C	3	3		9C	4	2	AC	6	2	EC	5	3				XH ← M, XL ← (M + 1)			7		8	
Decrement Index Reg	DEX														09	4	1	X ← X - 1						
Decrement Stack Pntr	DES														34	4	1	SP ← SP - 1						
Increment Index Reg	INX														08	4	1	X ← X + 1						
Increment Stack Pntr	INS														31	4	1	SP ← SP + 1						
Load Index Reg	LDX	CE	3	3		DE	4	2	EE	6	2	FE	5	3				M → XH, (M + 1) → XL			9			R
Load Stack Pntr	LDS	8E	3	3		9E	4	2	AE	6	2	BE	5	3				M → SPH, (M + 1) → SPL			9			R
Store Index Reg	STX					DF	5	2	EF	7	2	FF	6	3				XH → M, XL → (M + 1)			9			R
Store Stack Pntr	STS					9F	5	2	AF	7	2	BF	6	3				SPH → M, SPL → (M + 1)			9			R
Idx Reg → Stack Pntr	TXS														35	4	1	X ← SP						
Stack Pntr → Idx Reg	TSX														30	4	1	SP ← X						

TABLE 5 — JUMP AND BRANCH INSTRUCTIONS

														COND. CODE REG.										
		RELATIVE			INDEX			EXTND			IMPLIED													
OPERATIONS		MNEMONIC		OP	~	#	OP	~	#	OP	~	#	OP	~	#	BRANCH TEST			5	4	3	2	1	0
																	H	I	N	Z	V	C		
Branch Always	BRA	20	4	2													•	•	•	•	•	•		
Branch If Carry Clear	BCC	24	4	2													•	•	•	•	•	•		
Branch If Carry Set	BCS	25	4	2													•	•	•	•	•	•		
Branch If = Zero	BEQ	27	4	2													•	•	•	•	•	•		
Branch If ≥ Zero	BGE	2C	4	2													•	•	•	•	•	•		
Branch If > Zero	BGT	2E	4	2													•	•	•	•	•	•		
Branch If Higher	BHI	22	4	2													•	•	•	•	•	•		
Branch If ≤ Zero	BLE	2F	4	2													•	•	•	•	•	•		
Branch If Lower Or Same	BLS	23	4	2													•	•	•	•	•	•		
Branch If < Zero	BLT	2D	4	2													•	•	•	•	•	•		
Branch If Minus	BMI	2B	4	2													•	•	•	•	•	•		
Branch If Not Equal Zero	BNE	26	4	2													•	•	•	•	•	•		
Branch If Overflow Clear	BVC	28	4	2													•	•	•	•	•	•		
Branch If Overflow Set	BVS	29	4	2													•	•	•	•	•	•		
Branch If Plus	BPL	2A	4	2													•	•	•	•	•	•		
Branch To Subroutine	BSR	8D	8	2													•	•	•	•	•	•		
Jump	JMP				6E	4	2	7E	3	3						} See Special Operations	•	•	•	•	•	•		
Jump To Subroutine	JSR				AD	8	2	BD	9	3							•	•	•	•	•	•		
No Operation	NOP										01	2	1			Advances Prog. Cntr. Only	•	•	•	•	•	•		
Return From Interrupt	RTI										3B	10	1				•	•	•	•	•	•		
Return From Subroutine	RTS										39	5	1			} See Special Operations	•	•	•	•	•	•		
Software Interrupt	SWI										3F	12	1				•	•	•	•	•	•		
Wait for Interrupt*	WAI										3E	9	1			•	•	•	•	•	•			

*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

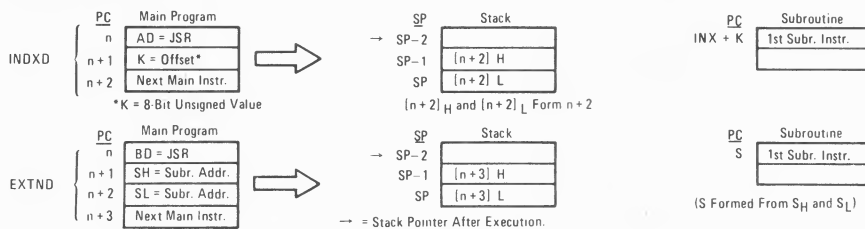


MOTOROLA Semiconductor Products Inc.

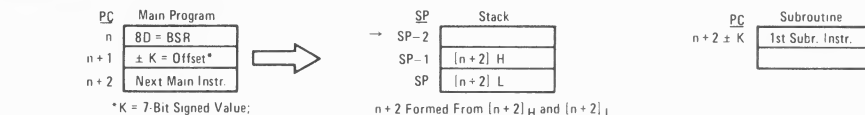
MC6800

SPECIAL OPERATIONS

JSR, JUMP TO SUBROUTINE:



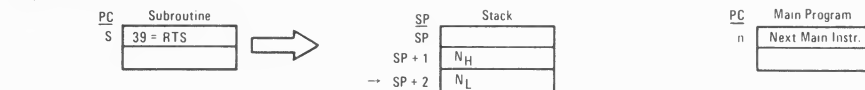
BSR, BRANCH TO SUBROUTINE:



JMP, JUMP:



RTS, RETURN FROM SUBROUTINE:



RTI, RETURN FROM INTERRUPT:

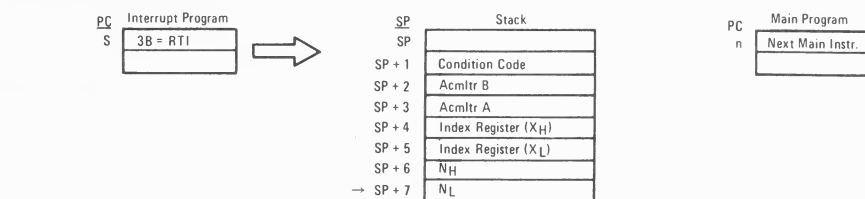


TABLE 6 — CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

OPERATIONS	MNEMONIC	IMPLIED			BOOLEAN OPERATION	COND. CODE REG.					
		OP	~	=		5	4	3	2	1	0
Clear Carry	CLC	0C	2	1	0 → C	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	0 → I	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	0 → V	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	1 → C	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	1 → I	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	1 → V	•	•	•	•	S	•
Acmltr A → CCR	TAP	06	2	1	A → CCR	12					
CCR → Acmltr A	TPA	07	2	1	CCR → A						

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- (Bit V) Test: Result = 10000000?
- (Bit C) Test: Result = 00000000?
- (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
- (Bit V) Test: Operand = 10000000 prior to execution?
- (Bit V) Test: Operand = 01111111 prior to execution?
- (Bit V) Test: Set equal to result of N ⊙ C after shift has occurred.
- (Bit N) Test: Sign bit of most significant (MS) byte = 1?
- (Bit V) Test: 2's complement overflow from subtraction of MS bytes?
- (Bit N) Test: Result less than zero? (Bit 15 = 1)
- (All) Load Condition Code Register from Stack. (See Special Operations)
- (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
- (All) Set according to the contents of Accumulator A.



MOTOROLA Semiconductor Products Inc.

MC6800

TABLE 7 — INSTRUCTION ADDRESSING MODES AND ASSOCIATED EXECUTION TIMES
(Times in Machine Cycles)

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied	Relative		(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied
ABA		•	•	•	•	•	•	•	INC	•	•	•	•	•	•	•
ADC	x	•	•	•	•	•	•	•	INS	•	•	•	•	•	•	•
ADD	x	•	•	•	•	•	•	•	INX	•	•	•	•	•	•	•
AND	x	•	•	•	•	•	•	•	JMP	•	•	•	•	•	•	•
ASL	•	•	•	•	•	•	•	•	JSR	•	•	•	•	•	•	•
ASR	•	•	•	•	•	•	•	•	LDA	x	•	•	•	•	•	•
BCC	•	•	•	•	•	•	•	•	LDS	•	•	•	•	•	•	•
BCS	•	•	•	•	•	•	•	•	LDX	•	•	•	•	•	•	•
BEA	•	•	•	•	•	•	•	•	LSR	•	•	•	•	•	•	•
BGE	•	•	•	•	•	•	•	•	NEG	•	•	•	•	•	•	•
BGT	•	•	•	•	•	•	•	•	NOP	•	•	•	•	•	•	•
BHI	•	•	•	•	•	•	•	•	ORA	x	•	•	•	•	•	•
BIT	x	•	•	•	•	•	•	•	PSH	•	•	•	•	•	•	•
BLE	•	•	•	•	•	•	•	•	PUL	•	•	•	•	•	•	•
BLS	•	•	•	•	•	•	•	•	ROL	•	•	•	•	•	•	•
BLT	•	•	•	•	•	•	•	•	ROR	•	•	•	•	•	•	•
BMI	•	•	•	•	•	•	•	•	RTI	•	•	•	•	•	•	•
BNE	•	•	•	•	•	•	•	•	RTS	•	•	•	•	•	•	•
BPL	•	•	•	•	•	•	•	•	SBA	•	•	•	•	•	•	•
BRA	•	•	•	•	•	•	•	•	SBC	x	•	•	•	•	•	•
BSR	•	•	•	•	•	•	•	•	SEC	•	•	•	•	•	•	•
BVC	•	•	•	•	•	•	•	•	SEI	•	•	•	•	•	•	•
BVS	•	•	•	•	•	•	•	•	SEV	•	•	•	•	•	•	•
CBA	•	•	•	•	•	•	•	•	STA	x	•	•	•	•	•	•
CLC	•	•	•	•	•	•	•	•	STS	•	•	•	•	•	•	•
CLI	•	•	•	•	•	•	•	•	STX	•	•	•	•	•	•	•
CLR	•	•	•	•	•	•	•	•	SUB	x	•	•	•	•	•	•
CLV	•	•	•	•	•	•	•	•	SWI	•	•	•	•	•	•	•
CMP	x	•	•	•	•	•	•	•	TAB	•	•	•	•	•	•	•
COM	•	•	•	•	•	•	•	•	TAP	•	•	•	•	•	•	•
CPX	•	•	•	•	•	•	•	•	TBA	•	•	•	•	•	•	•
DAA	•	•	•	•	•	•	•	•	TPA	•	•	•	•	•	•	•
DEC	•	•	•	•	•	•	•	•	TST	•	•	•	•	•	•	•
DES	•	•	•	•	•	•	•	•	TSX	•	•	•	•	•	•	•
DEX	•	•	•	•	•	•	•	•	TSX	•	•	•	•	•	•	•
EOR	x	•	•	•	•	•	•	•	WAI	•	•	•	•	•	•	•

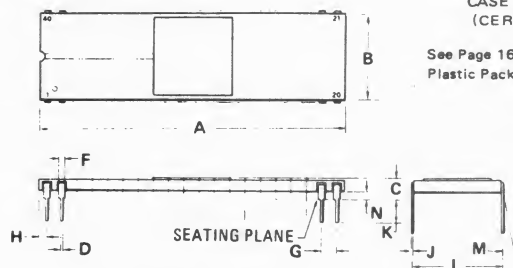
NOTE: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.

PIN ASSIGNMENT

1	O	Reset	40
2	VSS	TSO	39
3	φ1	N.C.	38
4	IRQ	φ2	37
5	VMA	DBE	36
6	NMI	N.C.	35
7	BA	R/W	34
8	VCC	D0	33
9	A0	D1	32
10	A1	D2	31
11	A2	D3	30
12	A3	D4	29
13	A4	D5	28
14	A5	D6	27
15	A6	D7	26
16	A7	A15	25
17	A8	A14	24
18	A9	A13	23
19	A10	A12	22
20	A11	VSS	21

PACKAGE DIMENSIONS
CASE 715-02
(CERAMIC)

See Page 165 for
Plastic Package dimensions.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.60	15.37	0.575	0.605
M		10°		10°
N	0.51	1.52	0.020	0.060

NOTE:
1. LEADS, TRUE POSITIONED WITHIN
0.25 mm (0.010) DIA (AT SEATING
PLANE), AT MAX. MAT'L
CONDITION.



MOTOROLA Semiconductor Products Inc.

MC6800
SUMMARY OF CYCLE BY CYCLE OPERATION

Table 8 provides a detailed description of the information present on the Address Bus, Data Bus, Valid Memory Address line (VMA), and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expected results during debug of both software and hard-

ware as the control program is executed. The information is categorized in groups according to Addressing Mode and Number of Cycles per instruction. (In general, instructions with the same Addressing Mode and Number of Cycles execute in the same manner; exceptions are indicated in the table.)

TABLE 8 — OPERATION SUMMARY

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus	
IMMEDIATE							
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Operand Data	
CPX LDS LDX		3	1	1	Op Code Address	1	Op Code
			2	1	Op Code Address + 1	1	Operand Data (High Order Byte)
	3		1	Op Code Address + 2	1	Operand Data (Low Order Byte)	
DIRECT							
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand	
		3	1	Address of Operand	1	Operand Data	
CPX LDS LDX	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand	
		3	1	Address of Operand	1	Operand Data (High Order Byte)	
		4	1	Operand Address + 1	1	Operand Data (Low Order Byte)	
STA	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Destination Address	
		3	0	Destination Address	1	Irrelevant Data (Note 1)	
		4	1	Destination Address	0	Data from Accumulator	
STS STX	5	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand	
		3	0	Address of Operand	1	Irrelevant Data (Note 1)	
		4	1	Address of Operand	0	Register Data (High Order Byte)	
		5	1	Address of Operand + 1	0	Register Data (Low Order Byte)	
INDEXED							
JMP	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Offset	
		3	0	Index Register	1	Irrelevant Data (Note 1)	
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)	
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	5	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Offset	
		3	0	Index Register	1	Irrelevant Data (Note 1)	
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)	
		5	1	Index Register Plus Offset	1	Operand Data	
CPX LDS LDX	6	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Offset	
		3	0	Index Register	1	Irrelevant Data (Note 1)	
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)	
		5	1	Index Register Plus Offset	1	Operand Data (High Order Byte)	
		6	1	Index Register Plus Offset + 1	1	Operand Data (Low Order Byte)	



MC6800

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INDEXED (Continued)						
STA	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	1	Index Register Plus Offset	1	Current Operand Data
		6	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		7	1/0 (Note 3)	Index Register Plus Offset	0	New Operand Data (Note 3)
STS STX	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data (High Order Byte)
		7	1	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
JSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer – 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer – 2	1	Irrelevant Data (Note 1)
		7	0	Index Register	1	Irrelevant Data (Note 1)
		8	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
EXTENDED						
JMP	3	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Jump Address (High Order Byte)
		3	1	Op Code Address + 2	1	Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data
CPX LDS LDX	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data (High Order Byte)
		5	1	Address of Operand + 1	1	Operand Data (Low Order Byte)
STA A STA B	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Destination Address (High Order Byte)
		3	1	Op Code Address + 2	1	Destination Address (Low Order Byte)
		4	0	Operand Destination Address	1	Irrelevant Data (Note 1)
		5	1	Operand Destination Address	0	Data from Accumulator
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Current Operand Data
		5	0	Address of Operand	1	Irrelevant Data (Note 1)
		6	1/0 (Note 3)	Address of Operand	0	New Operand Data (Note 3)

MC6800

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
EXTENDED (Continued)						
STS STX	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	0	Address of Operand	1	Irrelevant Data (Note 1)
		5	1	Address of Operand	0	Operand Data (High Order Byte)
		6	1	Address of Operand + 1	0	Operand Data (Low Order Byte)
JSR	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Subroutine (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
		4	1	Subroutine Starting Address	1	Op Code of Next Instruction
		5	1	Stack Pointer	0	Return Address (Low Order Byte)
		6	1	Stack Pointer – 1	0	Return Address (High Order Byte)
		7	0	Stack Pointer – 2	1	Irrelevant Data (Note 1)
		8	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		9	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
INHERENT						
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1 2	1 1	Op Code Address Op Code Address + 1	1 1	Op Code Op Code of Next Instruction
DES DEX INS INX	4	1 2 3 4	1 1 0 0	Op Code Address Op Code Address + 1 Previous Register Contents New Register Contents	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data (Note 1) Irrelevant Data (Note 1)
PSH	4	1 2 3 4	1 1 1 0	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer – 1	1 1 0 1	Op Code Op Code of Next Instruction Accumulator Data Accumulator Data
PUL	4	1 2 3 4	1 1 0 1	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer + 1	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data (Note 1) Operand Data from Stack
TSX	4	1 2 3 4	1 1 0 0	Op Code Address Op Code Address + 1 Stack Pointer New Index Register	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data (Note 1) Irrelevant Data (Note 1)
TXS	4	1 2 3 4	1 1 0 0	Op Code Address Op Code Address + 1 Index Register New Stack Pointer	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data Irrelevant Data
RTS	5	1 2 3 4 5	1 1 0 1 1	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer + 1 Stack Pointer + 2	1 1 1 1 1	Op Code Irrelevant Data (Note 2) Irrelevant Data (Note 1) Address of Next Instruction (High Order Byte) Address of Next Instruction (Low Order Byte)



MOTOROLA Semiconductor Products Inc.

MC6800

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INHERENT (Continued)						
WAI	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer — 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer — 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer — 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer — 4	0	Contents of Accumulator A
		8	1	Stack Pointer — 5	0	Contents of Accumulator B
		9	1	Stack Pointer — 6 (Note 4)	1	Contents of Cond. Code Register
RTI	10	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Contents of Cond. Code Register from Stack
		5	1	Stack Pointer + 2	1	Contents of Accumulator B from Stack
		6	1	Stack Pointer + 3	1	Contents of Accumulator A from Stack
		7	1	Stack Pointer + 4	1	Index Register from Stack (High Order Byte)
		8	1	Stack Pointer + 5	1	Index Register from Stack (Low Order Byte)
		9	1	Stack Pointer + 6	1	Next Instruction Address from Stack (High Order Byte)
		10	1	Stack Pointer + 7	1	Next Instruction Address from Stack (Low Order Byte)
SWI	12	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 1)
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer — 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer — 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer — 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer — 4	0	Contents of Accumulator A
		8	1	Stack Pointer — 5	0	Contents of Accumulator B
		9	1	Stack Pointer — 6	0	Contents of Cond. Code Register
		10	0	Stack Pointer — 7	1	Irrelevant Data (Note 1)
		11	1	Vector Address FFFA (Hex)	1	Address of Subroutine (High Order Byte)
		12	1	Vector Address FFFB (Hex)	1	Address of Subroutine (Low Order Byte)
RELATIVE						
BCC BHI BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMI BVS	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		4	0	Branch Address	1	Irrelevant Data (Note 1)
BSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer — 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer — 2	1	Irrelevant Data (Note 1)
		7	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		8	0	Subroutine Address	1	Irrelevant Data (Note 1)

Note 1. If device which is addressed during this cycle uses VMA, then the Data Bus will go to the high impedance three-state condition. Depending on bus capacitance, data from the previous cycle may be retained on the Data Bus.

Note 2. Data is ignored by the MPU.

Note 3. For TST, VMA = 0 and Operand data does not change.

Note 4. While the MPU is waiting for the interrupt, Bus Available will go high indicating the following states of the control lines: VMA is low; Address Bus, R/W, and Data Bus are all in the high impedance state.


MOTOROLA Semiconductor Products Inc.

**MOTOROLA**

SEMICONDUCTORS

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

Advance Information

MICROCOMPUTER UNIT (MCU)

The MC6801 MCU is an 8-bit microcomputer system which is compatible with the M6800 family of parts. The MC6801 MCU is object code compatible with the MC6800 with improved execution times of key instructions plus several new 16-bit and 8-bit instructions including an 8 X 8 unsigned multiply with 16-bit result. The MC6801 MCU can operate as a single chip microcomputer or be expanded to 65K words. The MC6801 MCU is TTL compatible and requires one +5.0 volt power supply. The MC6801 MCU has 2K bytes of ROM and 128 bytes of RAM on chip, Serial Communications Interface (S.C.I.), and parallel I/O as well as a three function 16-bit timer. Block diagram is shown in Figure 1. Features of the MC6801 include the following:

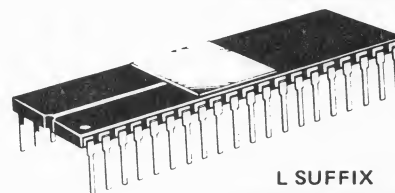
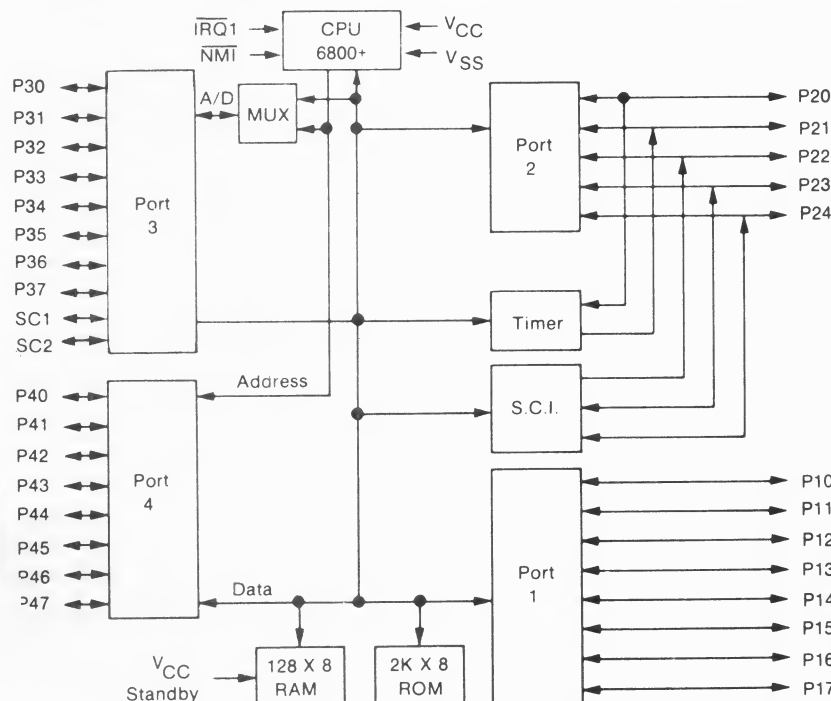
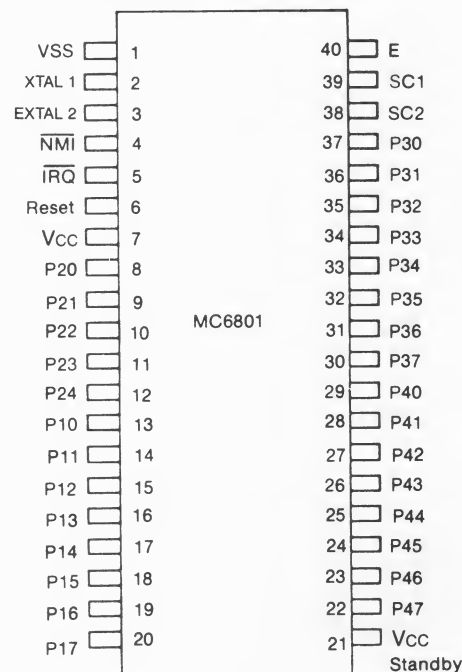
- Expanded M6800 Instruction Set
- 8 X 8 Multiply
- On-Chip Serial Communications Interface (S.C.I.)
- Object Code Compatible With The MC6800 MPU
- 16-Bit Timer
- Single Chip Or Expandable To 65K Words
- 2K Bytes Of ROM
- 128 Bytes Of RAM (64 Bytes Retainable On Power Down)
- 31 Parallel I/O Lines
- Internal Clock/Divided-By-Four
- TTL Compatible Inputs And Outputs
- Interrupt Capability
- External Clock/Divide-By-One Mask Option (MC6801E) And EPROM Versions MC68701 And MC68701E Available Soon.

MC6801

MOS

(N-CHANNEL, SILICON-GATE
DEPLETION LOAD)

MICROCOMPUTER

L SUFFIX
CERAMIC PACKAGE
CASE 715P SUFFIX
PLASTIC PACKAGE
CASE 711**FIGURE 1 - SINGLE-CHIP MICROCOMPUTER BLOCK DIAGRAM****FIGURE 2 — PIN ASSIGNMENT**

ELECTRICAL CHARACTERISTICS ($V_{CC} = 5.0V \pm 5\%$, $V_{SS} = 0$, $T_A = T_L$ to T_H unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Reset	V_{IH}	$V_{SS} + 2.0$ $V_{SS} + 4.0$	-	V_{CC} V_{CC}	Vdc
Input Low Voltage	V_{IL}	$V_{SS} - 0.3$	-	$V_{SS} + 0.8$	Vdc
Three-State (Off State) Input Current P10-P17 ($V_{in} = 0.4$ to 2.4 Vdc) P20-P24, P30-P37	I_{TSI} I_{TSI}	- -	2.0 2.0	10 10	μ Adc μ Adc
Output High Voltage All Outputs Except XTAL 1 and EXTERNAL 2 ($I_{Load} = -200 \mu$ Adc)	V_{OH}	$V_{SS} + 2.4$	-	-	Vdc
Output Low Voltage All Outputs Except XTAL 1 and EXTERNAL 2 ($I_{Load} = 1.6$ mAdc)	V_{OL}	-	-	$V_{SS} + 0.4$	Vdc
Power Dissipation	P_D	-	-	1200	mW
Capacitance ($V_{in} = 0$, $T_A = 25^\circ C$, $f = 1.0$ MHz) P10-P17, P20-P24, P40-P47 P30-P37 Reset SC1, SC2, IRQ	C_{in}	- - -	- - -	12.5 10 7.5	pF
Peripheral Data Setup Time (Figure 5)	t_{PDSU}	200	-	-	ns
Peripheral Data Hold Time (Figure 5)	t_{PDH}	0	-	-	ns
Delay Time, Enable negative transition to OS3 negative transition	t_{OSD1}	-	-	1.0	μ s
Delay Time, Enable negative transition to OS3 positive transition	t_{OSD2}	-	-	1.0	μ s
Delay Time, Enable negative transition to Peripheral Data Valid (Figure 6)	t_{PWD}	-	-	350	ns
Delay Time, Enable negative transition to Peripheral CMOS Data Valid ($V_{CC} - 30\% V_{CC}$, P20-P24 (Figure 6)	t_{CMOS}	-	-	2.0	μ s
Darlington Drive Current $V_O = 1.5$ Vdc P10-P17	I_{OH}	-1.0	-2.5	-10	mAdc
Standby Voltage (Not Operating)	V_{SBB}	4.00	-	5.25	Vdc
(Operating)	V_{SB}	4.75	-	5.25	Vdc

NOTE: The above electricals satisfy Ports 1 and 2 always, and Ports 3 and 4 in the single chip mode only.

BUS TIMING (Figure 9)

Characteristic	Symbol	Min	Typ	Max	Unit
Cycle Time	t_{CYC}	1000	-	-	ns
Address Strobe Pulse Width High	PW_{ASH}	220	-	-	ns
Address Strobe Rise Time	t_{ASR}	-	-	50	ns
Address Strobe Fall Time	t_{ASF}	-	-	50	ns
Address Strobe Delay Time	t_{ASD}	60	-	-	ns
Enable Rise Time	t_{ER}	-	-	50	ns
Enable Fall Time	t_{EF}	-	-	50	ns
Enable Pulse Width High Time	PW_{EH}	450	-	-	ns
Enable Pulse Width Low Time	PW_{EL}	450	-	-	ns
Address Strobe to Enable Delay Time	t_{ASED}	60	-	-	ns
Address Delay Time	t_{AD}	-	-	270	ns
Data Delay Write Time	t_{DDW}	-	-	225	ns
Data Set-up Time	t_{DSR}	100	-	-	ns
Hold Time } Read	t_{HR}	20	-	100	ns
} Write	t_{HW}	20	-	-	ns
Address Delay Time for Latch	t_{ADL}	-	-	200	ns
Address Hold Time for Latch	t_{AHL}	20	-	-	ns
Pulse Width	PW_0	370	370	-	ns
Address Hold Time	t_{AH}	20	-	-	ns
Total Up Time	t_{UT}	750	-	-	ns



MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	VCC	-0.3 to +7.0	Vdc
Input Voltage	V _{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	TA	0 to 70	°C
Storage Temperature Range	Tstg	-55 to +150	°C
Thermal Resistance Plastic Package	θ_{JA}	100	°C/W
Ceramic Package		50	

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit. For proper operation it is recommended that V_{in} and V_{out} be constrained to the range $V_{SS} \leq (V_{in} \text{ or } V_{out}) \leq V_{DD}$.

TABLE 1 — MODE AND PORT SUMMARY

MCU SIGNAL DESCRIPTION

This section gives a description of the MCU signals for the various modes. Figure 2 shows the general pin assignments for the signals. SC1 and SC2 are signals which vary with the mode that the chip is in. Table 1 gives a summary of their function.

MODE	PORT 1 Eight Lines	PORT 2 Five Lines	PORT 3 Eight Lines	PORT 4 Eight Lines	SC1	SC2
SINGLE CHIP	I/O	I/O	I/O	I/O	$\overline{IS3(I)}$	$\overline{OS3(O)}$
EXPANDED MUX	I/O	I/O	ADDRESS BUS (A0-A7) DATA BUS (D0-D7)	ADDRESS BUS* (A8-A15)	AS(O)	R/ \overline{W} (O)
EXPANDED NON-MUX	I/O	I/O	DATA BUS (D0-D7)	ADDRESS BUS* (A0-A7)	$\overline{IOS}(O)$	R/ \overline{W} (O)

*These lines can be substituted for I/O (Input Only) starting with the most significant address line.

I = Input

IS = Input Strobe

SC = Strobe Control

O = Output

OS = Output Strobe

AS = Address Strobe

R/ \overline{W} = Read/Write

IOS = I/O Select

READ/WRITE TIMING FOR PORTS 3 AND 4 (Figures 3-4)

Characteristic	Symbol	Min	Typ	Max	Unit
Address Delay	t _{AD}	-	-	270	ns
Peripheral Read Access Time t _{acc} = t _{ut} - (t _{AD} + t _{DSR})	t _{acc}	-	-	530	ns
Data Setup Time (Read)	t _{DSR}	100	-	-	ns
Input Data Hold Time	t _{HR}	10	-	-	ns
Output Data Hold Time	t _{HW}	20	-	-	ns
Address Hold Time (Address, R/W)	t _{AH}	20	-	-	ns
Data Delay Time (Write)	t _{DDW}	-	165	225	ns
Processor Controls					
Processor Control Setup Time	t _{PCS}	200	-	-	ns
Processor Control Rise and Fall Time (Measured between 0.8V and 2.0V)	t _{PCr} , t _{PCf}	-	-	100	ns
				100	

PORT 3 STROBE TIMING (Figures 7-8)

Characteristic	Symbol	Min	Typ	Max	Unit
Output Strobe Delay 1	t _{DSD1}	-	-	1.0	μs
Output Strobe Delay 2	t _{OSD2}	-	-	1.0	μs
Input Strobe Pulse Width	PW _{is}	200	-	-	ns
Input Data Hold Time	t _{IH}	20	-	-	ns
Input Data Setup Time	t _{IS}	100	-	-	ns



FIGURE 3 — READ DATA FROM MEMORY OR PERIPHERALS EXPANDED NON-MULTIPLEXED

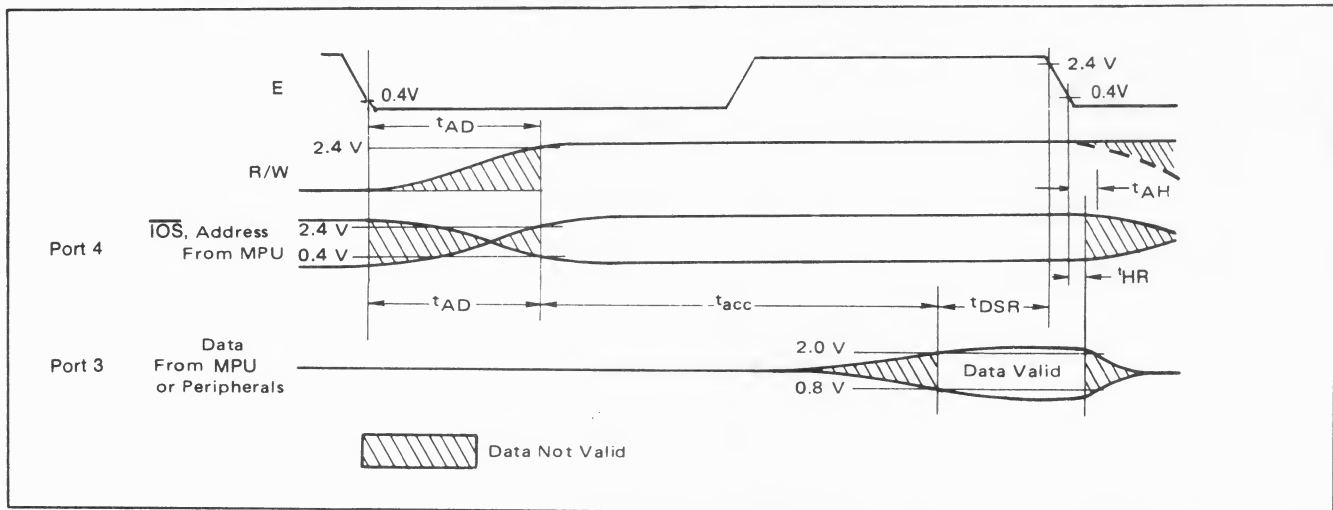
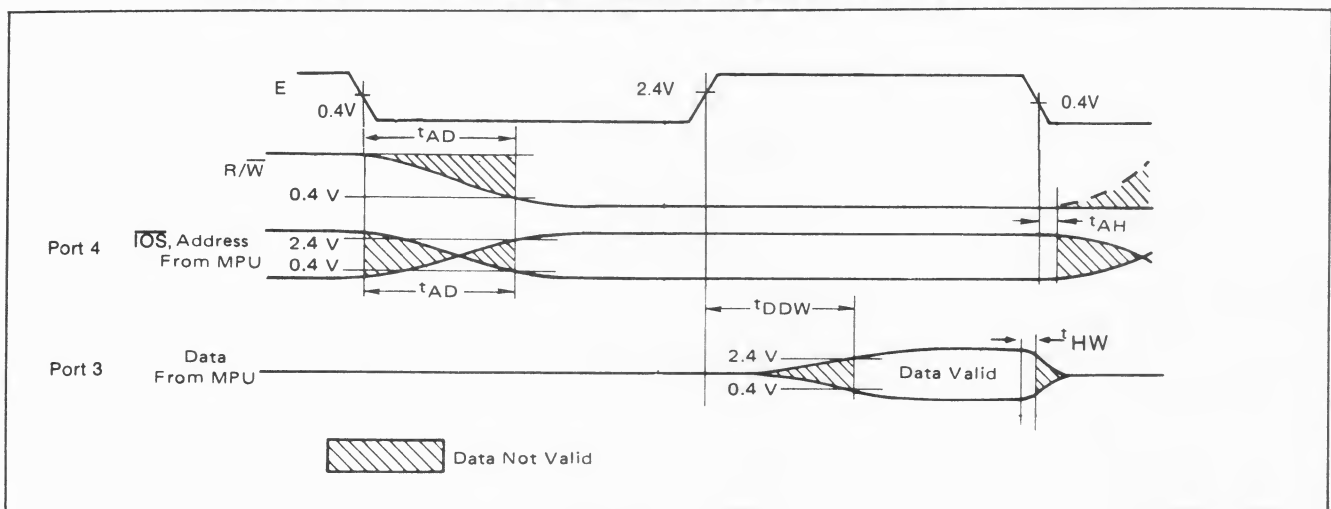


FIGURE 4 — WRITE DATA IN MEMORY OR PERIPHERALS EXPANDED NON-MULTIPLEXED



PORTS 1 AND 2, AND PORTS 3 AND 4 IN THE SINGLE CHIP MODE

FIGURE 5 — PERIPHERAL DATA SETUP AND HOLD TIMES (Read Mode)

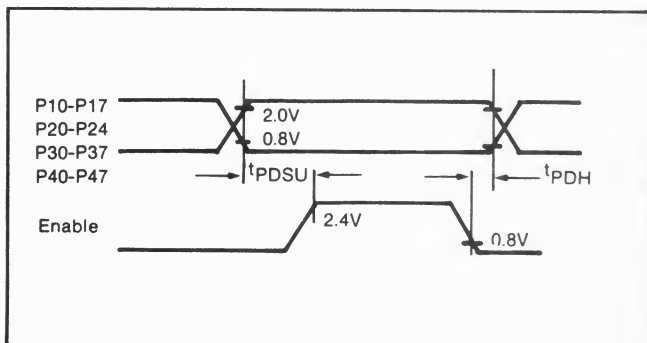


FIGURE 6 — PERIPHERAL CMOS DATA DELAY TIMES (Write Mode)

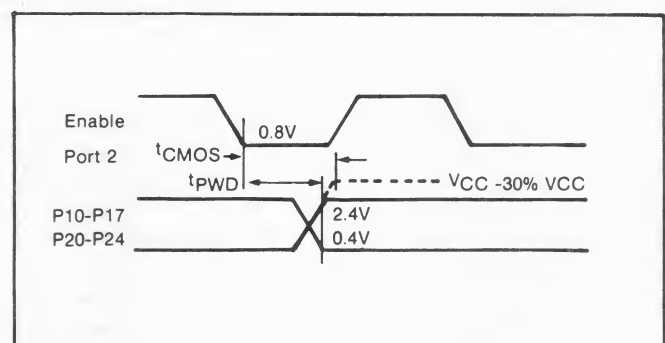


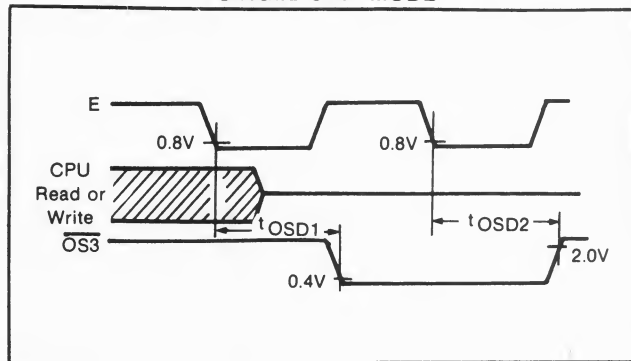
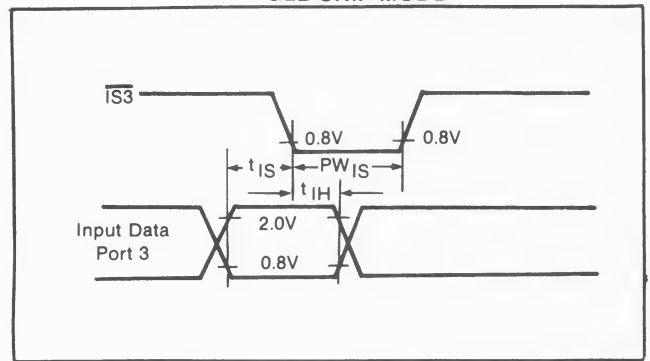
FIGURE 7 — OUTPUT STROBE TIMING —
SINGLE CHIP MODEFIGURE 8 — INPUT STROBE TIMING —
SINGLE CHIP MODE

FIGURE 9 — MULTIPLEXED BUS TIMING

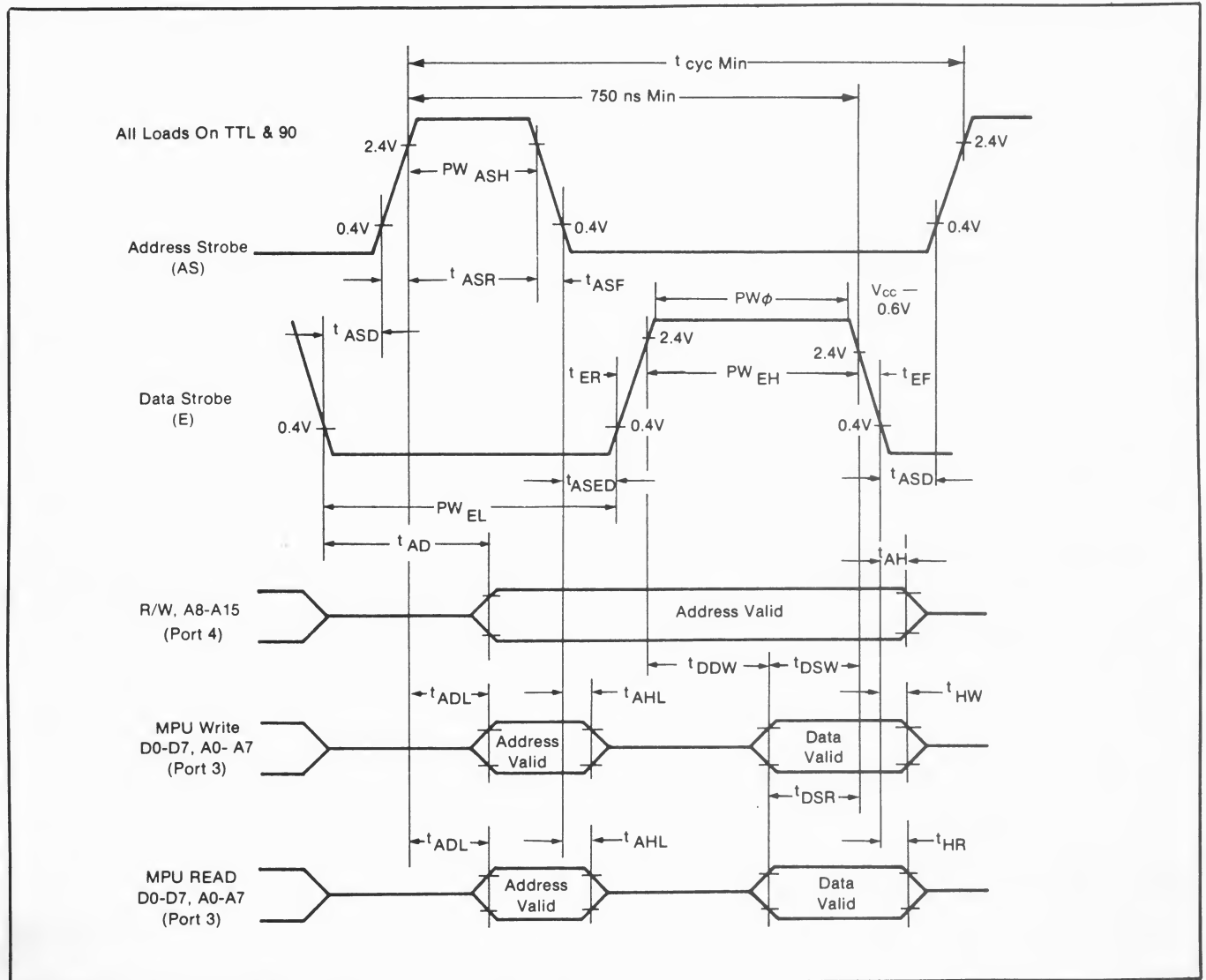


FIGURE 10—CMOS LOAD

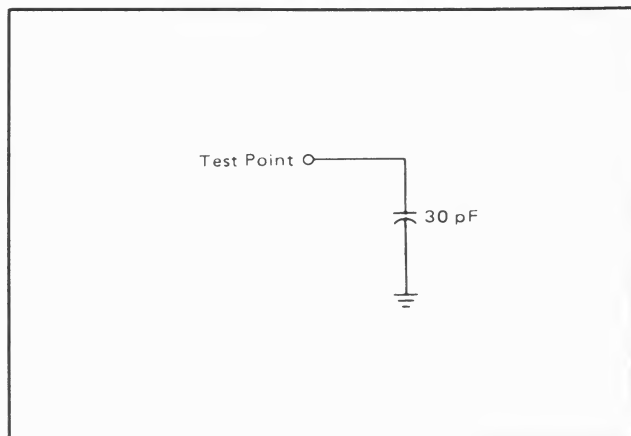
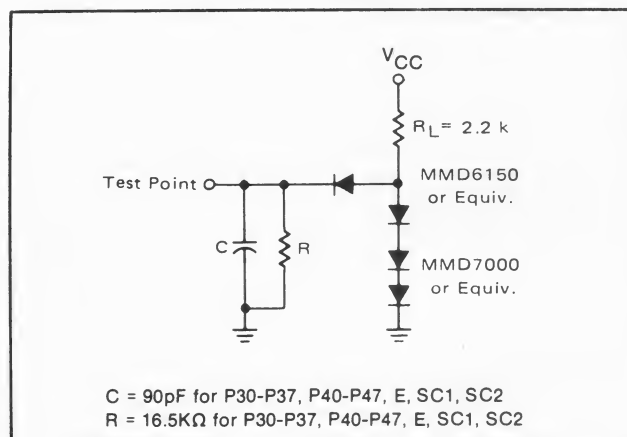
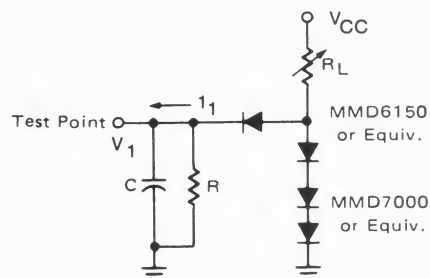
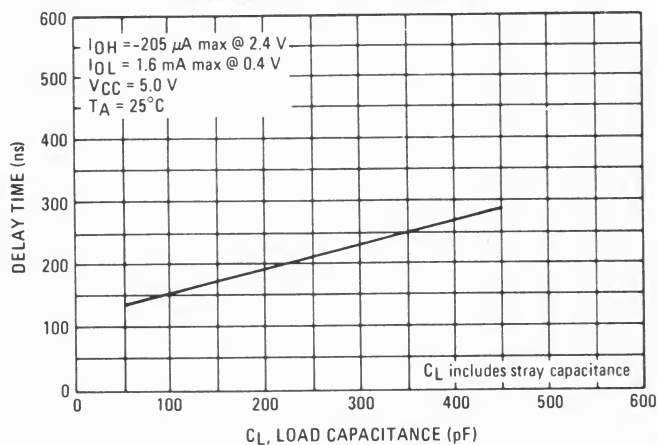
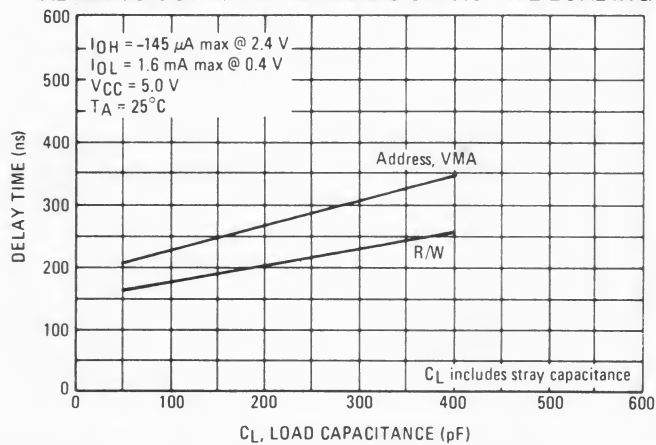


FIGURE 11 — BUS TIMING TEST LOAD AND PORTS 1, 3 AND 4 FOR SINGLE CHIP MODE

FIGURE 12 — TEST LOADS FOR PORT 1
Darlington Load
(P10-P17)

$C = 40\text{ R}_L$, $R = 12\text{ k}$
 Adjust R_L so that $I_1 = 3.2\text{ mA}$
 with $V_1 = 0.4\text{ V}$ and $V_{CC} = 5.25\text{ V}$

FIGURE 13 — TYPICAL DATA BUS OUTPUT DELAY
versus CAPACITIVE LOADINGFIGURE 14 — TYPICAL READ/WRITE, VMA AND
ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING

SIGNAL DESCRIPTIONS

Vcc and Vss

These two pins are used to supply power and ground to the chip. The voltage supplied will be +5 volts $\pm 5\%$.

XTAL 1 and XTAL 2

These connections are for a parallel resonant fundamental crystal, AT cut. Divide by 4 circuitry is included with the internal clock, so a 4 MHz crystal may be used to run the system at 1 MHz. The divide by 4 circuitry allows for use of the inexpensive 3.56 MHz Color TV crystal for non-time critical applications. Two 27 pF capacitors are needed from the two crystal pins to ground to insure reliable operation. XTAL2 may be driven by an external clock source at a 4 MHz rate to run at 1 MHz with a 40/60% duty cycle. It is not restricted to 4 MHz, as it will divide by 4 any frequency less than or equal to 4 MHz. XTAL1 must be grounded if an external clock is used. The following are the recommended crystal parameters:

AT = Cut Parallel Resonance Crystal
 $C_o = 7$ pF MAX
 FREQ = 4.0 MHz @ $C_L = 24$ pF
 $R_s = 50$ ohms MAX.
 Frequency Tolerance - $\pm 5\%$ to $\pm 0.02\%$
 The best E output "Worst Case Design"
 tolerance is $\pm 0.05\%$ (500 ppm) using
 A $\pm 0.02\%$ crystal.

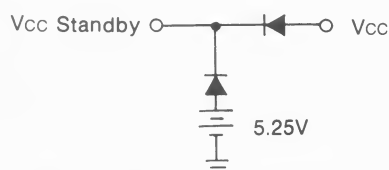
Vcc Standby

This pin will supply +5 volts $\pm 5\%$ to the standby RAM on the chip. The first 64 bytes of RAM will be maintained in the power down mode with 8 mA current max in the ROM version. The circuit of figure 15 can be utilized to assure that Vcc Standby does not go below VssB during power down.

To retain information in the RAM during power down the following procedure is necessary:

- 1) Write "0" into the RAM enable bit, RAM E. RAM E is bit 6 of the RAM Control Register at location \$0014. This disables the standby RAM, thereby protecting it at power down.
- 2) Keep Vcc Standby greater than VssB.

FIGURE 15—BATTERY BACKUP FOR Vcc STANDBY

**Reset**

This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. On power up, the reset must be held low for at least 20 ms. During operation, Reset, when brought low, must be held low at least 3 clock cycles.

When a high level is detected, the MPU does the following:

- a) All the higher order address lines will be forced high.
- b) I/O Port 2 bits, 2, 1, and 0 are latched into programmed control bits PC2, PC1 and PC0.
- c) The last two (FFFE, FFFF) locations in memory will be used to load the program addressed by the program counter.
- d) The interrupt mask bit is set, must be cleared before the MPU can recognize maskable interrupts.

Enable (E)

This supplies the external clock for the rest of the system when the internal oscillator is used. It is a single phase, TTL

compatible clock, and will be the divide by 4 result of the crystal frequency. It will drive one TTL load and 90 pF.

Non-Maskable Interrupt (NMI)

A low-going edge on this input requests that a non-maskable-interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the NMI signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

In response to an NMI interrupt, the Index Register, Program Counter, Accumulators, and Condition Code Register are stored on the stack. At the end of the sequence, a 16-bit address will be loaded that points to a vectoring address located in memory locations FFFC and FFFD. An address loaded at these locations causes the MPU to branch to a non-maskable interrupt service routine in memory.

A 3.3 k Ω external resistor to Vcc should be used for wire-OR and optimum control of interrupts.

Inputs \overline{IRQ} and \overline{NMI} are hardware interrupt lines that are sampled during E and will start the interrupt routine on the clock bar following the completion of an instruction.

Interrupt Request (\overline{IRQ})

This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further maskable interrupts may occur. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine in memory.

The IRQ requires a 3.3 k Ω external resistor to Vcc which should be used for wire-OR and optimum control of interrupts. Internal Interrupts will use an internal interrupt line (IRQ2). This interrupt will operate the same as IRQ except that it will use the vector address of FFF0 and FFF7. IRQ1 will have priority over IRQ2 if both occur at the same time. The Interrupt Mask Bit in the condition mode register masks both interrupts. (See Figure 25).

The following pins are available in the Single Chip Mode, and are associated with Port 3 only.

Input Strobe ($\overline{IS3}$) (SC1)

This sets an interrupt for the processor when the IS3 Enable bit is set. As shown in Figure 8 Input Strobe Timing, IS3 will fall T_{IS} minimum after data is valid on Port 3. If IS3 Enable is set in the I/O Port Control/Status Register, an interrupt will occur. If the latch enable bit in the I/O Control Status Register is set, this strobe will latch the input data from another device when that device has indicated that it has valid data.

Output Strobe ($\overline{OS3}$) (SC2)

This signal is used by the processor to strobe an external device, indicating valid data is on the I/O pins. The timing for the Output Strobe is shown in Figure 7. I/O Port Control/Status Register is discussed in the following section.



The following pins are available in the Expanded Modes.

Read/Write (R/W) (SC2)

This TTL compatible output signals the peripherals and memory devices whether the MPU is in a Read (high) or a Write (low) state. The normal standby state of this signal is Read (high). This output is capable of driving one TTL load and 90 pF.

I/O Strobe ($\overline{\text{IOS}}$) (SC1)

In the expanded non-multiplexed mode of operation, $\overline{\text{IOS}}$ internally decodes A9 through A15 as zero's and A8 as a one. This allows external access of the 256 locations from \$0100 to \$01FF. The timing diagrams are shown as figures 3 and 4.

Address Strobe (AS) (SC1)

In the expanded multiplexed mode of operation address strobe is output on this pin. This signal is used to latch the 8 LSB's of address which are multiplexed with data on Port 3. An 8-bit latch is utilized in conjunction with Address Strobe, as shown in figure 29, Expanded Multiplexed Mode. Address Strobe signals the latch when it is time to latch the address lines so the lines can become data bus lines during the E pulse. The timing for this signal is shown in the MC6801 Bus Timing Figure 9. This signal is also used to disable the address from the multiplexed bus allowing a deselection time, T_{ASD} before the data is enabled to the bus.

MC6801 PORTS

There are four I/O ports on the MC6801 MCU; three 8-bit ports and one 5-bit port. There are two control lines associated with one of the 8-bit ports. Each port has an associated write only Data Direction Register which allows each I/O line to be programmed to act as an input or an output.* A "1" in the corresponding Data Direction Register bit will cause that I/O line to be an output. A "0" in the corresponding Data Direction Register bit will cause that I/O line to be an input. There are four ports: Port 1, Port 2, Port 3, and Port 4. Their addresses and the addresses of their Data Direction registers are given in Table 2.

*The only exception is bit 1 of Port 2, which can either be data input or Timer output.

TABLE 2 — PORT AND DATA DIRECTION REGISTER ADDRESSES

Ports	Port Address	Data Direction Register Address
I/O Port 1	\$0002	\$0000
I/O Port 2	\$0003	\$0001
I/O Port 3	\$0006	\$0004
I/O Port 4	\$0007	\$0005

I/O Port 1

This is an 8-bit port whose individual bits may be defined as inputs or outputs by the corresponding bit in its data direction register. The 8 output buffers have three-state capability, allowing them to enter a high impedance state when the peripheral data lines are used as inputs. In order to be read properly, the voltage on the input lines must be greater than 2.0 volts for a logic "1" and less than 0.8 volt for a logic "0". As outputs, these lines are TTL compatible and may also be used as a source of up to 1 mA at 1.5 volts to directly drive a Darlington base. After Reset, the I/O lines are configured as inputs. In all three modes, Port 1 is always parallel I/O.

I/O Port 2

This port has five lines that may be defined as inputs or outputs by its data direction register. The 5 output buffers have three-state capability, allowing them to enter a high impedance state when used as an input. In order to be read properly, the voltage on the input lines must be greater than 2.0 volts for a logic "1" and less than 0.8 volt for a logic "0". As outputs, this port has no internal pullup resistors but will drive TTL inputs directly. For driving CMOS inputs, external pullup resistors are required. After Reset, the I/O lines are configured as inputs. Three pins on Port 2 (pins 10, 9 and 8 of the chip) are used to program the mode of operation during reset. The values of these pins at reset are latched into the three MSB's (bits 7, 6, and 5) of Port 2 which are read only. This is explained in the Mode Selection Section.

In all three modes, Port 2 can be configured as I/O and provides access to the Serial Communications Interface and the Timer. Bit 1 is the only pin restricted to data input or Timer output.

I/O Port 3

This is an 8-bit port that can be configured as I/O, a data bus, or an address bus multiplexed with the data bus — depending on the mode of operation hardware programmed by the user at reset. As a data bus, Port 3 is bi-directional. As an input for peripherals, it must be supplied regular TTL levels, that is, greater than 2.0 volts for a logic "1" and less than 0.8 volt for a logic "0".

Its TTL compatible three-state output buffers are capable of driving one TTL load and 90 pF. In the Expanded Modes, after reset, the data direction register is inhibited and data flow depends on the state of the R/W line. The input strobe (IS3) and the output strobe (OS3) used for handshaking are explained later.

In the three modes Port 3 assumes the following characteristics:

Single Chip Mode: Parallel Inputs/Outputs as programmed by its associated Data Direction Register. There are two control lines associated with this port in this mode, an input strobe and an output strobe, that can be used for handshaking. They are controlled by the I/O Port Control/Status Register explained at the end of this section.

Expanded Non-Multiplexed Mode: In this mode Port 3 becomes the data bus (D7-D0).

Expanded Multiplexed Mode: In this mode Port 3 becomes both the data bus (D7-D0) and lower bits of the address bus (A7-A0). An address strobe output is true when the address is on the port.

I/O PORT 3 CONTROL/STATUS REGISTER

	7	6	5	4	3	2	1	0
	IS3	IS3	X	OSS	LATCH	X	X	X
\$000F	FLAG	ENABLE			ENABLE			

Bit 0 Not used.

Bit 1 Not used.

Bit 2 Not used.

Bit 3 **Latch Enable.** This controls the input latch for I/O Port 3. If this bit is set high the input data will be latched with the falling edge of the Input Strobe, IS3. This bit is cleared by reset, or CPU Read Port 3.



Bit 4 (OSS) Output Strobe Select. This bit will select if the Output Strobe should be generated by a write to I/O Port 3 or a read of I/O Port 3. When this bit is cleared the strobe is generated by a read Port 3. When this bit is set the strobe is generated by a write Port 3.

Bit 5 Not used.

Bit 6 IS3 ENABLE. This bit will be the interrupt caused by IS3. When set to a low level the IS3 FLAG will be set by input strobe but the interrupt will not be generated. This bit is cleared by reset.

Bit 7 IS3 FLAG. This is a read only status bit that is set by the falling edge of the input strobe, IS3. It is cleared by a read of the Control/Status Register followed by a read or write of I/O Port 3. Reset will clear this bit.

I/O Port 4

This is an 8-bit port that can be configured as I/O or as address lines depending on the mode of operation. In order to be read properly, the voltage on the input lines must be greater than 2.0 volts for a logic "1" and less than 0.8 volt for a logic "0".

As outputs, each line is TTL compatible and can drive 1 TTL load and 90 pF. After reset, the lines are configured as inputs. To use the pins as addresses, therefore, they should be programmed as outputs. In the three modes, Port 4 assumes the following characteristics:

Single Chip Mode: Parallel Inputs/Outputs as programmed by its associated Data Direction Register.

Expanded Non-Multiplexed Mode: In this mode Port 4 is configured as the lower order address lines (A7-A0) by writing one's to the data direction register. When all eight address lines are not needed, the remaining lines, starting with the most significant bit, may be used as I/O (inputs only).

Expanded Multiplexed Mode: In this mode Port 4 is configured as the high order address lines (A15-A8) by writing one's to the data direction register. When all eight address lines are not needed, the remaining lines, starting with the most significant bit, may be used as I/O (inputs only).

MODE SELECTION

The mode of operation that 6801 will operate in after Reset is determined by hardware that the user must wire on pins 10, 9, and 8 of the chip. These pins are the three LSB's (I/O 2, I/O 1, and I/O 0

respectively) of Port 2. They are latched into programmed control bits PC2, PC1, and PC0 when reset goes high. I/O Port 2 Register is shown below.

	7	6	5	4	3	2	1	0
\$0003	PC2	PC1	PC0	I/O 4	I/O 3	I/O 2	I/O 1	I/O 0

An example of external hardware that could be used in the Expanded Non-Multiplexed Mode is given in Figure 16. In the Expanded Non-Multiplexed Mode, pins 10, 9 and 8 are programmed Hi, Lo, Hi respectively as shown.

Couplers between the pins on Port 2 and the peripherals attached may be required. If the lines go to devices which require signals at power up differing from the signals needed to program the 6801's mode, couplers are necessary.

The MC14066B can be used to provide this isolation between the peripheral device and the MCU during reset. Figure 17 shows the logic diagram and truth table for the MC14066B. It is bidirectional and requires no external logic to determine the direction of the information flow. The logic shown insures that the data on the peripheral will not change before it is latched into the MCU and the MCU has started the reset sequence.

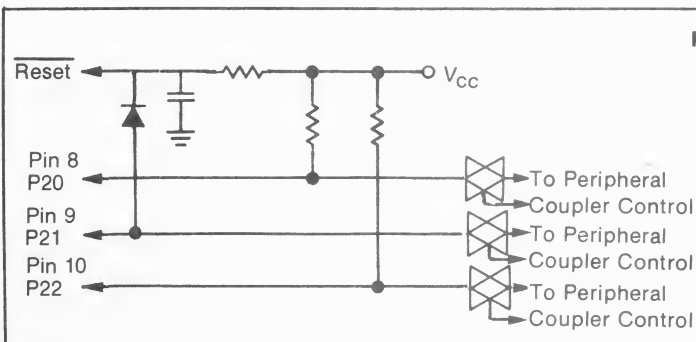


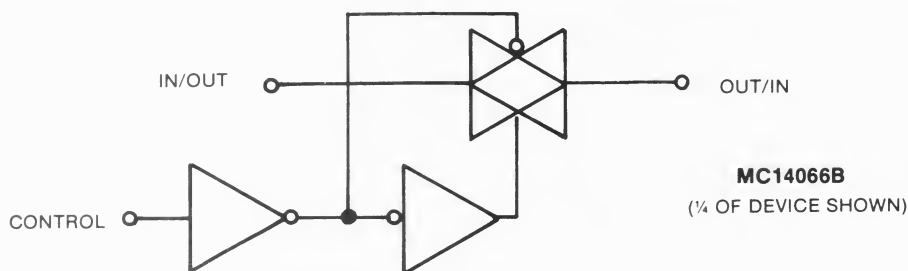
FIGURE 16 — DIODE CONFIGURATION FOR THE EXPANDED NON-MULTIPLEXED MODE

As bits 5, 6 and 7 of Port 2 are read only, the mode cannot be changed through software. The mode selections are shown in Table 3.

P20 refers to Port 2, bit 0.



**FIGURE 17—MC14066B QUAD ANALOG, SWITCH/
MULTIPLEXER IN A TYPICAL MC6801 CIRCUIT**



CONTROL	SWITCH
0	OFF
1	ON

V CONTROL	V _{in} TO V _{out} RESISTANCE
V _{SS}	> 10 ⁹ OHMS TYP.
V _{DD}	300 OHMS TYP.

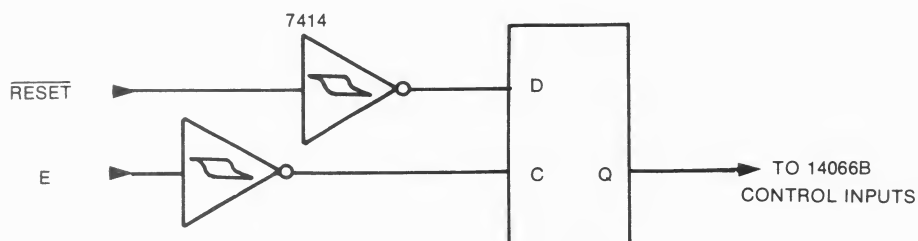


FIGURE 18 — MC6801 MCU SINGLE-CHIP MODE

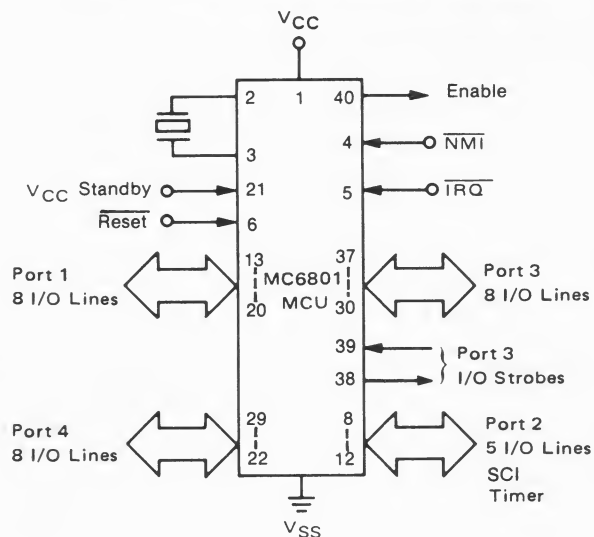
MC6801 BASIC MODES

The MC6801 is capable of operating in three basic modes; (1) Single Chip Mode, (2) Expanded Multiplexed Mode (compatible with M6800 peripheral family) (3) Expanded Non-Multiplexed Mode.

SINGLE CHIP MODE

In the Single Chip Mode the Ports are configured for I/O.

This is shown in Figure 18 the single Chip Mode. In this mode, Port 3 will have two associated control lines, an input strobe and an output strobe for handshaking data.



EXPANDED NON-MULTIPLEXED MODE

In this mode the MC6801 will directly address M6800 peripherals with no external logic. In this mode Port 3 becomes the data bus, Port 4 becomes the A7-A0 address bus or partial address and I/O (inputs only), Port 2 can be parallel I/O, serial I/O, Timer, or any combination thereof. Port 1 is parallel I/O

only. In this mode the MC6801 is expandable to 256 locations. The eight address lines associated with Port 4 may be substituted for I/O (inputs only) if a fewer number of address lines will satisfy the application. (See Figure 19).

FIGURE 19 — MC6801 MCU EXPANDED NON-MULTIPLEXED MODE

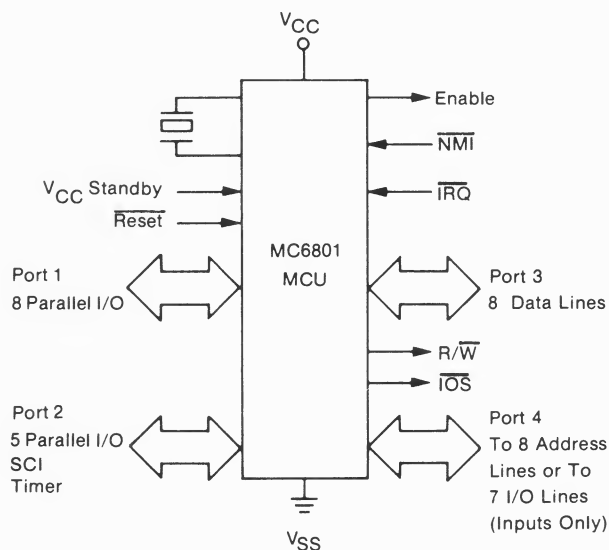
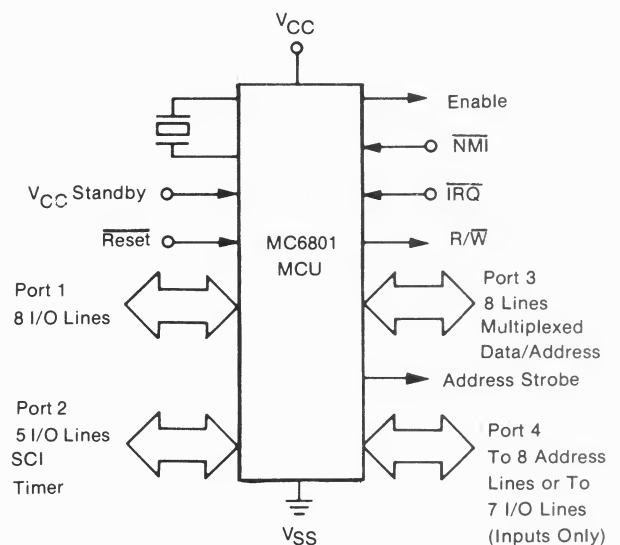


FIGURE 20 — MC6801 MCU EXPANDED MULTIPLEXED MODE



EXPANDED MULTIPLEXED MODE

In this mode Port 4 becomes higher order address lines with an alternative of substituting some of the address lines for I/O (inputs only). Port 3 is the data bus multiplexed with the lower order address lines differentiated by an output called Address Strobe. Port 2 is 5 lines of Parallel I/O, SCI, Timer, or any combination thereof. Port 1 is 8 Parallel I/O lines. In this mode it is expandable to 65K words. (See Figure 20).



TABLE 3 — MODE SELECTS

MODE		PROGRAM CONTROL			ROM	RAM	INTERRUPT VECTORS	BUS
7	SINGLE CHIP	Hi	Hi	Hi	I	I	I	I
6	EXPANDED MULTIPLEXED	Hi	Hi	Lo	I	I	I	Ep/M
5	EXPANDED NON-MULTIPLEXED	Hi	Lo	Hi	I	I	I	Ep
4	SINGLE CHIP TEST	Hi	Lo	Lo	I(2)	I(1)	I	I
3	64K ADDRESS I/O	Lo	Hi	Hi	E	E	E	Ep/M
2	PORTS 3 & 4 EXTERNAL	Lo	Hi	Lo	E	I	E	Ep/M
1		Lo	Lo	Hi	I	I	E	Ep/M
0	TEST-DATA OUTPUTTED FROM ROM & RAM TO I/O PORT 3	Lo	Lo	Lo	I	I	I*	Ep/M

E — EXTERNAL all vectors are external
 I — INTERNAL
 Ep — EXPANDED
 M — MULTIPLEXED

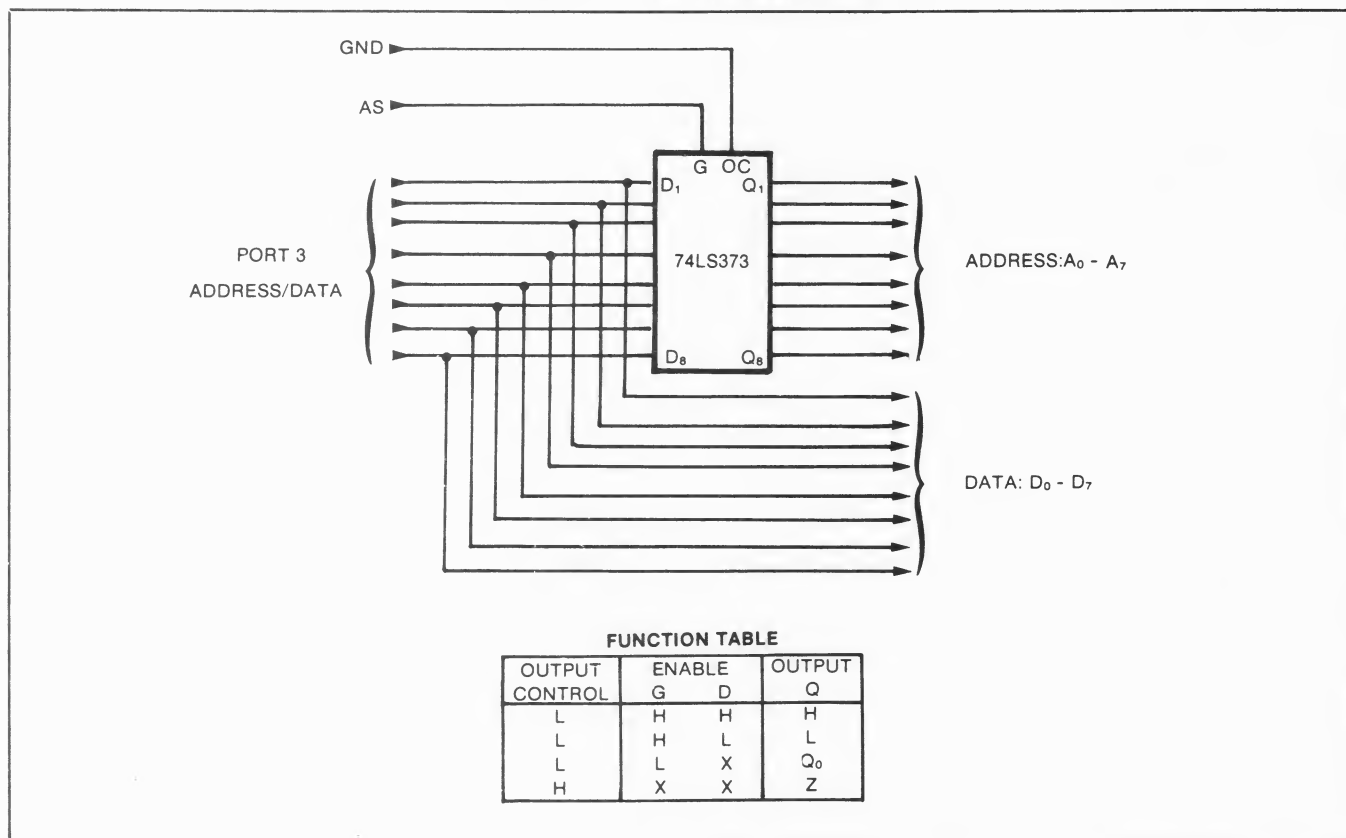
* First two addresses read from external after reset
 (1) Address for RAM XX80-XXFF
 (2) ROM disabled

Lower order Address Bus Latches

Since the data bus is multiplexed with the lower order address bus in Port 3, latches are required to latch those address bits. The SN74LS373 Transparent octal D-type latch

can be used with the MC6801 to latch the least significant address byte. Figure 21 shows how to connect the latch to the MC6801. The output control to the LS373 may be connected to ground.

FIGURE 21 — LATCH CONNECTION



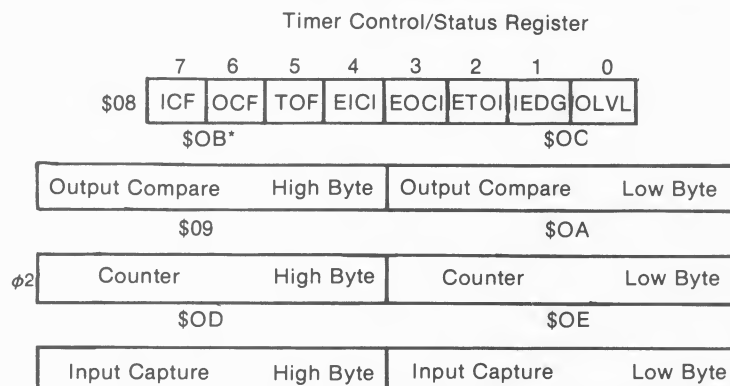
PROGRAMMABLE TIMER

The MC6801 contains an on-chip 16-bit programmable timer which may be used to perform measurements on an input waveform while independently generating an output waveform. Pulse widths for both input and output signals may vary from a few microseconds to many seconds. The timer hardware consists of

- an 8-bit control and status register,
- a 16-bit free running counter,
- a 16-bit output compare register, and
- a 16-bit input capture register

A block diagram of the timer registers is shown in Figure 22.

FIGURE 22 — BLOCK DIAGRAM OF TIMER REGISTERS

**Free Running Counter (\$0009:000A)**

The key element in the programmable timer is a 16-bit free running counter which is driven to increasing values by the MPU ϕ . The counter value may be read by the MPU software at any time. The counter is cleared to zero on RESET and may be considered a read-only register with one exception. Any MPU write to the counter's address (\$09) will always result in a preset value of \$FFF8 being loaded into the counter regardless of the value involved in the write. This preset feature is intended for testing operation of the part, but may be of value in some applications.

Output Compare Register (\$000B:000C)

The Output Compare Register is a 16-bit read/write register which is used to control an output waveform. The contents of this register are constantly compared with the current value of the free running counter. When a match is found, a flag is set (OCF) in the Timer Control and Status Register (TCSR) and the current value of the Output Level bit (OLVL) in the TCSR is clocked to the output level register. Providing the Data Direction Register for Port 2, Bit 1 contains a "1" (output), the output level register value will appear on the pin for Port 2 Bit 1. The values in the Output Compare Register and Output level bit may then be changed to control the output level on the next compare value. The Output Compare Register is set to \$FFFF during RESET. The Compare function is inhibited for one cycle following a write to the high byte of the Output Compare Register to insure a valid 16-bit value is in the register before a compare is made.

Input Capture Register (\$000D:000E)

The Input Capture Register is a 16-bit read-only register used to store the current value of the free running counter when the proper transition of an external input signal occurs. The input transition change required to trigger the counter transfer is controlled by the Input Edge bit (IEDG) in the TCSR. The Data Direction Register bit for Port 2 Bit 0, should* be clear (zero) in order to gate in the external input signal to the edge detect unit in the timer.

*With Port 2 Bit 0 configured as an output and set to "1", the external input will still be seen by the edge detect unit.

Timer Control and Status Register (TCSR) (\$0008)

The Timer Control and Status Register consists of an 8-bit register of which all 8 bits are readable but only the low order 5 bits may be written. The upper three bits contain read-only timer status information and indicate that:

- a proper transition has taken place on the input pin with a subsequent transfer of the current counter value to the input capture register,
- a match has been found between the value in the free running counter and the output compare register, and
- when \$0000 is in the free running counter.

Each of the flags may be enabled onto the MC6801 internal bus (IRQ2) with an individual Enable bit in the TCSR. If the I-bit in the MC6801 Condition Code register has been cleared, a priority vectored interrupt will occur corresponding to the flag bit(s) set. A description for each bit follows:



- Bit 0 **OLVL** Output Level — This value is clocked to the output level register on an output compare. If the DDR for Port 2 bit 1 is set, the value will appear on the output pin.
- Bit 1 **IEDG** Input Edge — This bit controls which transition of an input will trigger a transfer of the counter to the input capture register. The DDR for Port 2 Bit 0 must be clear for this function to operate. IEDG = 0 Transfer takes place on a negative (high-to-low transition). IEDG = 1 Transfer takes place on a positive edge (low-to-high transition).
- Bit 2 **ETOI** Enable Timer Overflow Interrupt — When **set**, this bit enables IRQ2 to occur on the internal bus for a TOF interrupt; when **clear** the interrupt is inhibited.
- Bit 3 **EOCI** Enable Output Compare Interrupt — When **set**, this bit enables IRQ2 to appear on the internal bus for an input capture interrupt; when **clear** the interrupt is inhibited.
- Bit 4 **EICI** Enable Input Capture Interrupt — When **set**, this bit enables IRQ2 to occur on the internal bus for an input capture interrupt; when **clear** the interrupt is inhibited.
- Bit 5 **TOF** Timer Overflow Flag — This read-only bit is **set** when the counter contains \$0000. It is **cleared** by a read of the TCSR (with TOF set) followed by an MPU read of the Counter (\$09).
- Bit 6 **OCF** Output Compare Flag — This read-only bit is **set** when a match is found between the output compare register and the free running counter. It is **cleared** by a read of the TCSR (with OCF set) followed by an MPU write to the output compare register (\$0B or \$0C).
- Bit 7 **ICF** Input Capture Flag — This read-only status bit is **set** by a proper transition on the input to the edge detect unit; it is **cleared** by a read of the TCSR (with ICF set) followed by an MPU read of the Input Capture Register (\$0D).

SERIAL COMMUNICATIONS INTERFACE

The MC6801 contains a full-duplex asynchronous serial communications interface (SCI) on board. Two serial data formats (standard mark/space (NRZ) or Bi-phase) are provided at several different data rates. The controller comprises a transmitter and a receiver which operate independently of each other but in the same data format and at the same data rate. Both transmitter and receiver communicate with the MPU via the data bus and with the outside world via pins 2, 3, and 4 of Port 2. The hardware, software, and registers are explained in the following paragraphs.

Wake-Up Feature

In a typical multi-processor application, the software protocol will usually contain a destination address in the initial byte(s) of the message. In order to permit non-selected MPU's to ignore the remainder of the message, a wake-up feature is included whereby all further interrupt processing may be optionally inhibited until the beginning of the next message. When the next message appears, the hardware re-enables (or "wakes-up") the for the next message. The "wake-up" is automatically triggered by a string of ten consecutive 1's which indicates an idle transmit line. The software protocol must provide for the short idle period between any two consecutive messages.

Programmable Options

The following features of the MC6801 serial I/O section are programmable:

- format — standard mark/space (NRZ) or Bi-phase
- clock — external or internal
- baud rate — one of 4 per given MPU ϕ 2 clock frequency or external clock X8 input
 - wake-up feature — enabled or disabled
 - interrupt requests — enabled or masked individually for transmitter and receiver data registers
 - clock output — internal clock enabled or disabled to Port 2 (Bit 2)
 - Port 2 (bits 3 and 4) — dedicated or not dedicated to serial I/O individually for transmitter and receiver.

Serial Communications Hardware

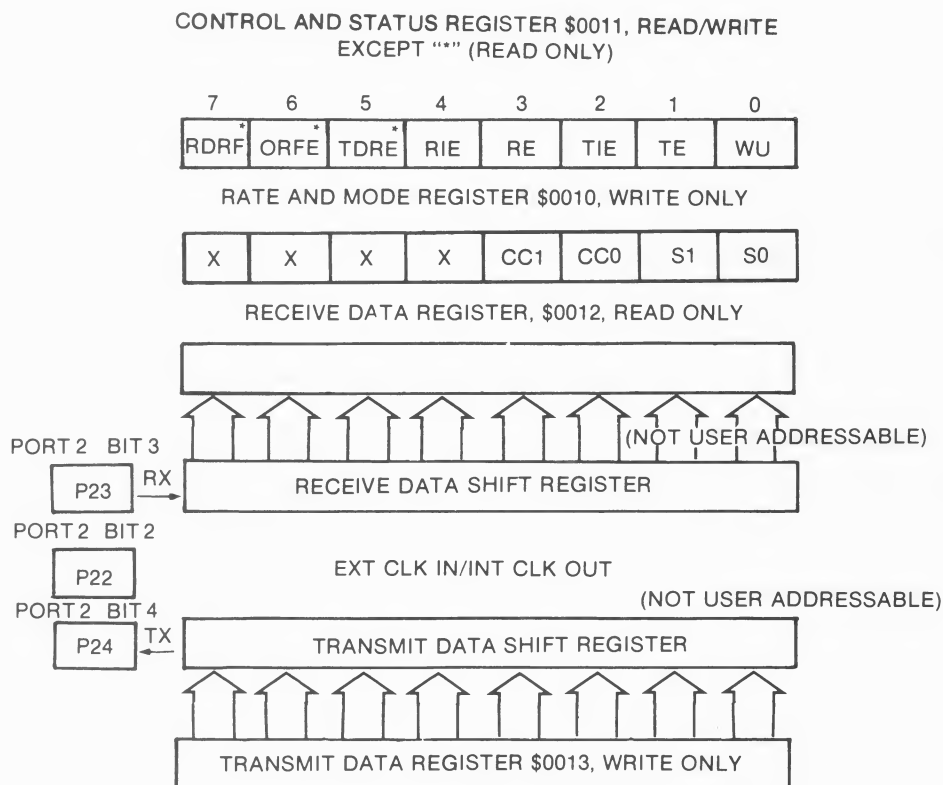
The serial communications hardware is controlled by 4 registers as shown in Figure 23. The registers include:

- an 8-bit control and status register
- a 4-bit rate and mode control register (write only)
- an 8-bit read only receive data register and
- an 8-bit write only transmit data register.

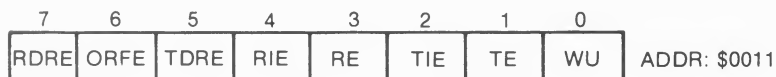
In addition to the four registers, the serial I/O section utilizes bit 3 (serial input) and bit 4 (serial output) or Port 2. Bit 2 of Port 2 is utilized if the internal-clock-out or external-clock-in options are selected.



FIGURE 23 — SERIAL I/O REGISTERS

**Transmit/Receive Control and Status (TRCS) Register**

The TRCS register consists of an 8-bit register of which all 8 bits may be read while only bits 0-4 may be written. The register is initialized to \$20 on RESET. The bits in the TRCS register are defined as follows:



- Bit 0 **WU** "Wake-up" on Next Message — set by MC6801 software cleared by hardware on receipt of ten consecutive 1's.
- Bit 1 **TE** Transmit Enable — set by MC6801/MC68701 to produce preamble of nine consecutive 1's and to enable gating of transmitter output to Port 2, bit 4 regardless of the DDR value corresponding to this bit; when clear, serial I/O has no effect on Port 2 bit 4.
- Bit 2 **TIE** Transmit Interrupt Enable — when set, will permit an $\overline{\text{IRQ2}}$ interrupt to occur when bit 5 (TDRE) is set; when clear, the TDRE value is masked from the bus.
- Bit 3 **RE** Receiver Enable — when set, gates Port 2 bit 3 to input of receiver regardless of DDR value for this bit; when clear, serial I/O has no effect on Port 2 bit 3.
- Bit 4 **RIE** Receiver Interrupt Enable — when set, will permit an $\overline{\text{IRQ2}}$ interrupt to occur when bit 7 (RDRF) or bit 6 (OR) is set; when clear, the interrupt is masked.

- Bit 5 **TDRE** Transmit Data Register Empty — set by hardware when a transfer is made from the transmit data register to the output shift register. The TDRE bit is cleared by reading the status register, then writing a new byte into the transmit data register, TDRE is initialized to 1 by $\overline{\text{RESET}}$.
- Bit 6 **ORFE** Over-Run-Framing Error — set by hardware when an overrun or framing error occurs (receive only). An overrun is defined as a new byte received with last byte still in Data Register/Buffer. A framing error has occurred when the byte boundaries in bit stream are not synchronized to bit counter. The ORFE bit is cleared by reading the status register, then reading the Receive Data Register, or by $\overline{\text{RESET}}$.
- Bit 7 **RDRF** Receiver Data Register Full — Set by hardware when a transfer from the input shift register to the receiver data register is made. The RDRF bit is cleared by reading the status register, then reading the Receive Data Register, or by $\overline{\text{RESET}}$.

Rate and Mode Control Register

The Rate and Mode Control register controls the following serial I/O variables:

- Baud rate
- format
- clocking source, and
- Port 2 bit 2 configuration

The register consists of 4 bits all of which are write-only and cleared on $\overline{\text{RESET}}$. The 4 bits in the register may be considered as a pair of 2-bit fields. The two low order bits control the bit rate for internal clocking and the remaining two bits control the format and clock select logic. The register definition is as follows:

7	6	5	4	3	2	1	0	
X	X	X	X	CC1	CC0	S1	S0	ADDR:\$0010

- Bit 0 **S0** Speed Select — These bits select the Baud rate for the internal clock. The four rates which may be selected are a function of the MPU ϕ_2 clock frequency. Table 4 lists the available Baud rates.
- Bit 1 **S1**
- Bit 2 **CC0** Clock Control and Format Select — this 2-bit field controls the format and clock select logic. Table 5 defines the bit field.
- Bit 3 **CC1**



TABLE 4 — SCI INTERNAL BAUD RATES

S1,S0	XTAL	4.0 MHz	4.9152 MHz	2.5476 MHz
	$\phi 2$	1.0 MHz	1.2288 MHz	0.6144 MHz
00	$\phi 2 \div 16$	62.5k Bits/s	76.8k Bits/s	38.4k Bits/s
01	$\phi 2 \div 128$	7,812.5 Bits/s	9,600 Bits/s	4,800 Bits/s
10	$\phi 2 \div 1024$	976.6 Bits/s	1,200 Bits/s	600 Bits/s
11	$\phi 2 \div 4096$	244.1 Bits/s	300 Bits/s	150 Bits/s

TABLE 5 — BIT FIELD

CC1, CC0	Format	Clock Source	Port 2 Bit 2	Port 2 Bit 3	Port 2 Bit 4
00	Bi-Phase	Internal	Not Used	**	**
01	NRZ	Internal	Not Used	**	**
10	NRZ	Internal	Output*	Serial Input	Serial Output
11	NRZ	External	Input	Serial Input	Serial Output

*Clock output is available regardless of values for bits RE and TE.

**Bit 3 is used for serial input if RE = "1" in TRCS; bit 4 is used for serial output if TE = "1" in TRCS.

Internally Generated Clock

If the user wishes for the serial I/O to furnish a clock, the following requirements are applicable:

- the values of RE and TE are immaterial.
- the values of RE and TE are immaterial.
- CC1, CC0 must be set to 10
- the maximum clock rate will be $\phi \div 16$.
- the clock will be at 1X the bit rate and will have a rising edge at mid-bit.

Externally Generated Clock

If the user wishes to provide an external clock for the serial I/O, the following requirements are applicable:

- the CC1, CC0, field in the Rate and Mode Control Register must be set to 11,
- the external clock must be set to 8 times (X8) the desired baud rate and
- the maximum external clock frequency is 1.3 MHz.



SERIAL OPERATIONS

The serial I/O hardware should be initialized by the MC6801 software prior to operation. This sequence will normally consist of:

- writing the desired operation control bits to the Rate and Mode Control Register and
- writing the desired operational control bits in the Transmit/Receive Control and Status Register.

The Transmitter Enable (TE) and Receiver Enable (RE) bits may be left set for dedicated operations.

Transmit Operations

The transmit operation is enabled by the TE bit in the Transmit/Receive Control and Status Register. This bit when set, gates the output of the serial transmit shift register to Port 2 Bit 4 and takes unconditional control over the Data Direction Register value for Port 2, Bit 4.

Following a **RESET**, the user should configure both the Rate and Mode Control Register and the Transmit/Receive Control and Status Register for desired operation. Setting the TE bit during this procedure initiates the serial output by first transmitting a ten-bit preamble of 1's. Following the preamble, internal synchronization is established and the transmitter section is ready for operation.

At this point one of two situations exist:

- a) if the Transmit Data Register is empty (TDRE = 1), a continuous string of ones will be sent indicating an idle line, or
- b) if data has been loaded into the Transmit Data Register (TDRE = 0), the word is transferred to the output shift register and transmission of the data word will begin.

During the transfer itself, the 0 start bit is first transmitted. Then the 8 data bits (beginning with bit 0) followed by the stop bit, are transmitted. When the Transmitter Data Register has been emptied, the hardware sets the TDRE flag bit.

If the MC6801 fails to respond to the flag within the proper time, (TDRE is still set when the next normal transfer from the parallel data register to the serial output register should occur) then a 1 will be sent (instead of a 0) at "Start" bit time, followed by more 1's until more data is supplied to the data register. No 0's will be sent while TDRE remains a 1.

The Bi-phase mode operates as described above except that the serial output toggles each bit time, and on 1/2 bit times when a 1 is sent.

Receive Operation

The receive operation is enabled by the RE bit which gates in the serial input through Port 2 Bit 3. The receiver section operation is conditioned by the contents of the Transmit/Receive Control and Status Register and the Rate and Mode Control Register.

The receiver bit interval is divided into 8 sub-intervals for internal synchronization. In the standard, non-Bi-phase mode, the received bit stream is synchronized by the first 0 (space) encountered.

The approximate center of each bit time is strobed during the next 10 bits. If the tenth bit is not a 1 (stop bit) a framing error is assumed, and bit ORFE is set. If the tenth bit is a 1, the data is transferred to the Receiver Data Register, and interrupt flag RDRF is set. If RDRF is still set at the next tenth bit time, ORFE will be set, indicating an over-run has occurred. When the MC6801 responds to either flag (RDRF or ORFE) by reading the status register followed by reading the Data Register, RDRF (or ORFE) will be cleared.

RAM CONTROL REGISTER

This register, which is addressed at \$0014, gives status information about the standby RAM. A 0 in the RAM enable bit (RAM E) will disable the standby RAM, thereby protecting it at power down if V_{CC} is held greater than V_{SB} volts, as explained previously in the signal description for V_{CC} Standby.

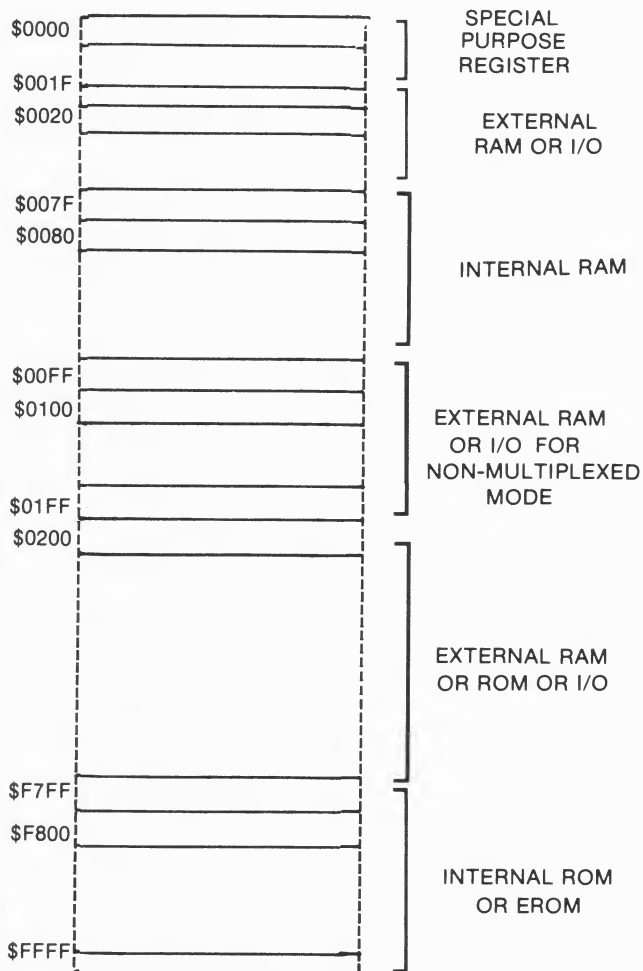
\$0014	STANDBY BIT						
	RAME	X	X	X	X	X	X

- Bit 0 Not Used.
- Bit 1 Not Used.
- Bit 2 Not used.
- Bit 3 Not used.
- Bit 4 Not used.
- Bit 5 Not used.
- Bit 6 The RAM ENABLE control bit allows the user the ability to disable the standby RAM. This bit is set to a logic "one" by reset which enables the standby RAM and can be written to one or zero under program control. When the RAM is disabled, logic "zero", data is read from external memory.
- Bit 7 The STANDBY BIT of the control register, \$0014, is cleared when the standby voltage is removed. This bit is a read/write status flag that the user can read which indicates that the standby RAM voltage has been applied, and the data in the standby RAM is valid.



FIGURE 24 — MEMORY MAP

The MC6801 provides up to 65k bytes of memory for program and/or data storage. The memory map is shown in Figure 24.



Locations \$0020 through \$007F access external RAM or I/O. Internal RAM is accessed at \$0080 through \$00FF. The RAM may be alternately selected by mask programming at location \$A080. However, if the user desires to access external RAM at those locations he may do so by clearing the RAM ENABLE control bit of the RAM Control Register. In this way an extra 128 bytes of external RAM are available. The first 64 bytes of the 128 bytes of on-chip RAM are provided with a separate power supply. This will maintain the 64 bytes of RAM in the power down mode as explained in the pin description for Vcc Standby.

Locations \$0100 through \$01FF are available in the Expanded Non-Multiplexed Mode. The eight address lines of Port 4 make

TABLE 6 — SPECIAL REGISTERS

The first 32 bytes are for the special purpose registers as shown in Table 6.

Hex Address	Register
00	Data Direction 1
01	Data Direction 2
02	I/O Port 1
03	I/O Port 2
04	Data Direction 3
05	Data Direction 4
06	I/O Port 3
07	I/O Port 4
08	TCSR
09	Counter High Byte
0A	Counter Low Byte
0B	Output Compare High Byte
0C	Output Compare Low Byte
0D	Input Capture High Byte
0E	Input Capture Low Byte
0F	I/O Port 3 C/S Register
10	Serial Rate and Mode Register
11	Serial Control and Status Register
12	Serial Receiver Data Register
13	Serial Transmit Data Register
14	RAM/EROM Control Register
15-1F	Reserved

FIGURE 25 — MEMORY MAP FOR INTERRUPT VECTORS

	Vector		Description
	MS	LS	
Highest Priority	FFFE, FFFF	Restart	
	FFFC, FFDD	Non-Maskable Interrupt	
	FFFA, FFFB	Software Interrupt	
	FFF8, FFF9	IRQ1/Interrupt Strobe 3	
	FFF6, FFF7	IRQ2/Timer Input Capture	
	FFF4, FFF5	IRQ2/Timer Output Compare	
Lowest Priority	FFF2, FFF3	IRQ2/Timer Overflow	
	FFF0, FFF1	IRQ2/Serial I/O Interrupt	

this 256 word expandability possible. Those not needed for address lines can be used as input lines instead.

The full range of addresses available to the user is in the Expanded Multiplexed Mode. Locations \$0200 through \$F7FF can be used as external RAM, external ROM, or I/O. Any higher order bits not required for addressing can be used as I/O as in the Expanded Non-Multiplexed Mode.

The internal ROM is located at \$F800 through \$FFFF. The decoder for the ROM may be mask programmed on A12, and A13 as zeros or one's to provide for \$C800, \$D800, \$E800 for the ROM address. A12 and A13 may also be don't care in this decoder. The primary address for the ROM will be \$F800.



GENERAL DESCRIPTION OF INSTRUCTION SET

The MC6801 is upward object code compatible with the MC6800 as it implements the full M6800 instruction set. The execution times of key instructions have been reduced to increase throughput. In addition, new instructions have been added; these include 16-bit operations and a hardware multiply.

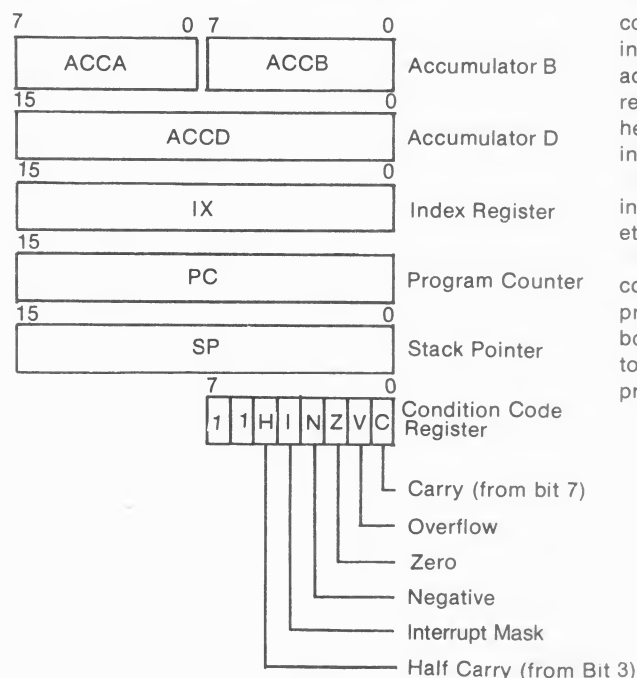
Included in the instruction set section are the following:

- MPU Programming Model (Figure 26)
- Addressing modes
- Accumulator and memory instructions — Table 7
- New instructions
- Index register and stack manipulations — Table 8
- Jump and branch instructions — Table 9
- Special operations — Figure 27
- Condition code register manipulation instructions — Table 10
- Instruction Execution times in machine cycles — Table 11
- Summary of cycle by cycle operation — Table 12

MPU PROGRAMMING MODEL

The programming model for the MC6801 is shown in Figure 26. The double (D) accumulator is physically the same as the A Accumulator concatenated with the B Accumulator so that any operation using accumulator D will destroy information in A and B.

FIGURE 26 — MCU PROGRAMMING MODEL



MPU ADDRESSING MODES

The MC6801 eight-bit microcomputer unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 11 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 4 MHz, these times would be microseconds.

Accumulator (ACCX) Addressing — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

Immediate Addressing — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MCU addresses this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

Direct Addressing — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

Extended Addressing — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

Indexed Addressing — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MCU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

Implied Addressing — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

Relative Addressing — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.



TABLE 7—ACCUMULATOR & MEMORY INSTRUCTIONS

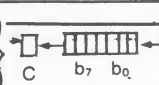

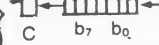

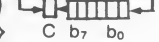
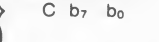

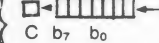
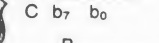
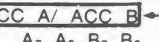

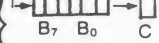
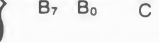

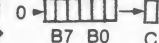
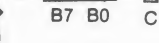
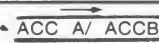
ACCUMULATOR AND MEMORY		ADDRESSING MODES																					
		IMMED.		DIRECT		INDEX		EXTEND		INHERENT		Boolean/Arithmetic Operation											
Operations	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	5	4	3	2	1	0	
Add	ADDA	8B	2	2	9B	3	2	AB	4	2	BB	4	3				A + M → A	↑	•	↑	↑	↑	↑
	ADDB	CB	2	2	DB	3	2	EB	4	2	FB	4	3				B + M → B	↑	•	↑	↑	↑	↑
Add Double	ADDD	C3	4	3	D3	5	2	E3	6	2	F3	6	3				A:B+M:M+1 → A:B	•	•	↑	↑	↑	↑
Add Accumulators	ABA													1B	2	1	A + B → A	↑	•	↑	↑	↑	↑
Add With Carry	ADCA	89	2	2	99	3	2	A9	4	2	B9	4	3				A + M + C → A		•	↑	↑	↑	↑
	ADCB	C9	2	2	D9	3	2	E9	4	2	F9	4	3				B + M + C → B	•	•	↑	↑	↑	↑
AND	ANDA	84	2	2	94	3	2	A4	4	2	B4	4	3				A M → A	•	•	↑	↑	R	•
	ANDB	C4	2	2	D4	3	2	E4	4	2	F4	4	3				B M → B	•	•	↑	↑	R	•
Bit Test	BIT A	85	2	2	95	3	2	A5	4	2	B5	4	3				A M	•	•	↑	↑	R	•
	BIT B	C5	2	2	D5	3	2	E5	4	2	F5	4	3				B M	•	•	↑	↑	R	•
Clear	CLR							6F	6	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	8I	2	2	9I	3	2	A1	4	2	B1	4	3				A - M	•	•	↑	↑	↑	↑
	CMPB	C1	2	2	D1	3	2	E1	4	2	F1	4	3				B - M	•	•	↑	↑	↑	↑
Compare Accumulators	CBA													1I	2	1	A - B	•	•	↑	↑	↑	↑
Complement, 1's	COM							63	6	2	73	6	3				M → M	•	•	↑	↑	R	S
	COMA													43	2	1	A → A	•	•	↑	↑	R	S
	COMB													53	2	1	B → B	•	•	↑	↑	R	S
Complement, 2's	NEG							60	6	2	70	6	3				0C - M → M	•	•	↑	↑	①	②
(Negate)	NEGA													40	2	1	00 - A → A	•	•	↑	↑	①	②
NEGB														50	2	1	00 - B → B	•	•	↑	↑	①	②
Decimal Adjust, A	DAA													19	2	1	Converts binary add of BCD characters into BCD format	•	•	↑	↑	↑	③
Decrement	DEC							6A	6	2	7A	6	3				M - 1 → M	•	•	↑	↑	④	•
	DECA													4A	2	1	A - 1 → A	•	•	↑	↑	④	•
	DECB													5A	2	1	B - 1 → B	•	•	↑	↑	④	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	4	2	B8	4	3				A ⊕ M → A	•	•	↑	↑	R	•
	EORB	C8	2	2	D8	3	2	E8	4	2	F8	4	3				B ⊕ M → B	•	•	↑	↑	R	•
Increment	INC							6C	6	2	7C	6	3				M + 1 → M	•	•	↑	↑	⑤	•
	INCA													4C	2	1	A + 1 → A	•	•	↑	↑	⑤	•
	INCB													5C	2	1	B + 1 → B	•	•	↑	↑	⑤	•
Load Accumulator	LDA A	86	2	2	96	3	2	A6	4	2	B6	4	3				M → A	•	•	↑	↑	R	•
	LDA B	C6	2	2	D6	3	2	E6	4	2	F6	4	3				M → B	•	•	↑	↑	R	•
Load Double Accumulator	LDAD	CC	3	3	DC	4	2	EC	5	2	FC	5	3				M → A M + 1 → B	•	•	↑	↑	R	•

The Condition Code Register notes are listed after Table 10.

(Continued)



TABLE 7 — Continued

ACCUMULATOR AND MEMORY		ADDRESSING MODES																					
		IMMED.			DIRECT			INDEX			EXTEND			INHERENT			5	4	3	2	1	0	
Operations	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	Boolean/ Arithmetic Operation	H	I	N	Z	V	C
Multiply Unsigned	MUL													3D	10	1	$A \times B \rightarrow A:B$	•	•	•	•	•	⑬
OR, Inclusive	ORAA	8A	2	2	9A	3	2	AA	4	2	BA	4	3				$A + M \rightarrow A$	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	4	2	FA	4	3				$B + M \rightarrow B$	•	•		↕	R	•
Push Data	PSHA													36	3	1	$A \rightarrow M_{sp}, SP - 1 \rightarrow SP$	•	•	•	•	•	•
	PSHB													37	3	1	$B \rightarrow M_{sp}, SP - 1 \rightarrow SP$	•	•	•	•	•	•
Pull Data	PULA													32	4	1	$SP + 1 \rightarrow SP, M_{sp} \rightarrow A$	•	•	•	•	•	•
	PULB													33	4	1	$SP + 1 \rightarrow SP, M_{sp} \rightarrow B$	•	•	•	•	•	•
Rotate Left	ROL							69	6	2	79	6	3					•	•	↕	↕	⑥	↕
	ROLA													49	2	1		•	•	↕	↕	⑥	↕
	ROLB													59	2	1		•	•	↕	↕	⑥	↕
Rotate Right	ROR							66	6	2	76	6	3					•	•	↕	↕	⑥	↕
	RORA													46	2	1		•	•	↕	↕	⑥	↕
	RORB													56	2	1		•	•	↕	↕	⑥	↕
Shift Left Arithmetic	ASL							68	6	2	78	6	3					•	•	↕	↕	⑥	↕
	ASLA													48	2	1		•	•	↕	↕	⑥	↕
	ASLB													58	2	1		•	•	↕	↕	⑥	↕
Double Shift Left, Arithmetic	ASLD													05	3	1		•	•	↕	↕	⑥	↕
Shift Right Arithmetic	ASR							67	6	2	77	6	3					•	•	↕	↕	⑥	↕
	ASRA													47	2	1		•	•	↕	↕	⑥	↕
	ASRB													57	2	1		•	•	↕	↕	⑥	↕
Shift Right, Logical	LSR							64	6	2	74	6	3					•	•	↕	↕	⑥	↕
	LSRA													44	2	1		•	•	↕	↕	⑥	↕
	LSRB													54	2	1		•	•	↕	↕	⑥	↕
Double Shift Right Logical	LSRD													04	3	1		•	•	R	↕	⑥	↕
Store Accumulator	STAA				97	3	2	A7	4	2	B7	4	3				$A \rightarrow M$	•	•	↕	↕	R	•
	STAB				D7	3	2	E7	4	2	F7	4	3				$B \rightarrow M$	•	•	↕	↕	R	•
Store Double Accumulator	STAD				DD	4	2	ED	5	2	FD	5	3				$A \rightarrow M$ $B \rightarrow M + 1$	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	4	2	B0	4	3				$A - M \rightarrow A$	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	4	2	F0	4	3				$B - M \rightarrow B$	•	•	↕	↕	↕	↕
Double Subtract	SUBD	83	4	3	93	5	2	A3	6	2	B3	6	3				$A:B - M:M + 1 \rightarrow A:B$	•	•	↕	↕	↕	↕
Subtract Accumulators	SBA													10	2	1	$A - B \rightarrow A$	•	•	↕	↕	↕	↕
Subtract With Carry	SBCA	82	2	2	92	3	2	A2	4	2	B2	4	3				$A - M - C \rightarrow A$	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	4	2	F2	4	3				$B - M - C \rightarrow B$	•	•	↕	↕	↕	↕
Transfer Accumulators	TAB													16	2	1	$A \rightarrow B$	•	•	↕	↕	R	•
	TBA													17	2	1	$B \rightarrow A$	•	•	↕	↕	R	•
Test Zero or Minus	TST							6D	6	2	7D	6	3				$M - 00$	•	•	↕	↕	RR	R
	TSTB													5D	2	1	$B - 00$	•	•	↕	↕	R	R

The Condition Code Register notes are listed after Table 10.



ADDED INSTRUCTIONS

In addition to the existing M6800 Instruction Set, the following new instructions are incorporated in the MC6801 Microcomputer.

ABX	Adds the 8-bit unsigned accumulator B to the 16-bit X-Register taking into account the possible carry out of the low order byte of the X-Register.	$IX \leftarrow IX + ACCB$
ADDD	Adds the double precision ACCD* to the double precision value M:M+1 and places the results in ACCD.	$ACCD \leftarrow (ACCD) + (M:M+1)$
ASLD	Shifts all bits of ACCAB one place to the left. Bit 0 is loaded with zero. The C bit is loaded from the most significant bit of ACCD.	
LDD	Loads the contents of double precision memory location into the double accumulator A:B. The condition codes are set according to the data.	$ACCD \leftarrow (M:M+1)$
LSRD	Shifts all bits of ACCD one place to the right. Bit 15 is loaded with zero. The C bit is loaded from the least significant bit to ACCD.	
MUL	Multiplies the 8 bits in accumulator A with the 8 bits in accumulator B to obtain a 16-bit unsigned number in A:B. ACCA contains MSB of result.	$ACCD \leftarrow ACCA * ACCB$
PSHX	The contents of the index register is pushed onto the stack at the address contained in the stack pointer. The stack pointer is decremented by 2.	$\downarrow (IXL), SP \leftarrow (SP) - 1$ $\downarrow (IXL), SP \leftarrow (SP) - 1$
PULX	The index register is pulled from the stack beginning at the current address contained in the stack pointer +1. The stack pointer is incremented by 2 in total.	$SP \leftarrow (SP) + 1; IXH$ $SP \leftarrow (SP) + 1; IHL$
STD	Stores the contents of double accumulator A:B in memory. The contents of ACCD remain unchanged.	$M:M + 1 \leftarrow (ACCD)$
SUBD	Subtracts the contents of M:M + 1 from the contents of double accumulator AB and places the result in ACCD.	$ACCB \leftarrow (ACCD) - (M:M + 1)$

*ACCD is the 16 bit register (A:B) formed by concatenating the A and B accumulators. The A-accumulator is the most significant byte.

TABLE 8 — INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

POINTER OPERATIONS		MNEMONIC		IMMED			DIRECT			INDEX			EXTND			IMPLIED			BOOLEAN/ARITHMETIC OPERATION	COND. CODE REG.					
				OP		#	OP		#	OP		#	OP		#	OP		#		5	4	3	2	1	0
				OP	~		OP	~		OP	~		OP	~		OP	~			OP	~	H	I	N	Z
Compare Index Reg	CPX	8C	4	3			9C	5	2	AC	6	2	BC	6	3				$X_H \leftarrow M, X_L \leftarrow (M + 1)$	•	•	⑦	↑	⑧	•
Decrement Index Reg	DEX															09	3	1	$X \leftarrow X - 1$	•	•	•	↑	•	•
Decrement Stack Pntr	DES															34	3	1	$SP \leftarrow SP - 1$	•	•	•	•	•	•
Increment Index Reg	INX															08	3	1	$X \leftarrow X + 1$	•	•	•	↑	•	•
Increment Stack Pntr	INS															31	3	1	$SP \leftarrow SP + 1$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3			DE	4	2	EE	5	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	⑨	↑	R	•
Load Stack Pntr	LDS	8E	3	3			9E	4	2	AE	5	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	⑨	↑	R	•
Store Index Reg	STX						DF	4	2	EF	5	2	FF	5	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	⑨	↑	R	•
Store Stack Pntr	STS						9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	⑨	↑	R	•
Index Reg → Stack Pntr	TXS															35	3	1	$X \leftarrow X - 1$	•	•	•	•	•	•
Stack Pntr → Index Reg	TSX															30	3	1	$SP \leftarrow SP + 1$	•	•	•	•	•	•
Add	ABX															3A	3	1	$B \leftarrow B + X$	•	•	•	•	•	•
Push Data	PSHX															3C	4	1	$X_L \rightarrow M_{SP}, SP \leftarrow SP - 1$	•	•	•	•	•	•
Pull Data	PULX																		$X_H \rightarrow H_{SP}, SP \leftarrow SP - 1$	•	•	•	•	•	•
																			$SP \leftarrow SP + 1, M_{SP} \rightarrow X_H$						
																			$SP \leftarrow SP + 1, M_{SP} \rightarrow X_L$						

The Condition Code Register notes are listed after Table 10.



TABLE 9 — JUMP AND BRANCH INSTRUCTIONS

														COND. CODE REG.							
OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			IMPLIED			BRANCH TEST	5	4	3	2	1	0	
		OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C	
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•	
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•	
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•	
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•	
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	•	•	•	•	•	•	
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	•	•	•	•	•	•	
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•	
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	•	•	•	•	•	•	
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•	
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	•	•	•	•	•	•	
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•	
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•	
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•	
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•	
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•	
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•	
Jump	JMP				6E	4	2	7E	3	3				} See Special Operations	•	•	•	•	•	•	
Jump To Subroutine	JSR				AD	8	2	8D	9	3					•	•	•	•	•	•	•
No Operation	NOP										01	2	1	} Advances Prog. Cntr. Only	•	•	•	•	•	•	
Return From Interrupt	RTI										3B	10	1		•	•	•	•	•	•	•
Return From Subroutine	RTS										39	5	1	} See Special Operations	•	•	•	•	•	•	
Software Interrupt	SWI										3F	12	1		•	•	•	•	•	•	•
Wait for Interrupt*	WAI										3E	9	1		•	•	•	•	•	•	•

TABLE 10 — CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

						COND. CODE REG.										
OPERATIONS	MNEMONIC	IMPLIED			BOOLEAN OPERATION	5	4	3	2	1	0					
		OP	~	=		H	I	N	Z	V	C					
Clear Carry	CLC	0C	2	1	0 → C	●	●	●	●	●	R					
Clear Interrupt Mask	CLI	0E	2	1	0 → I	●	R	●	●	●	●					
Clear Overflow	CLV	0A	2	1	0 → V	●	●	●	●	R	●					
Set Carry	SEC	0D	2	1	1 → C	●	●	●	●	●	S					
Set Interrupt Mask	SEI	0F	2	1	1 → I	●	S	●	●	●	●					
Set Overflow	SEV	0B	2	1	1 → V	●	●	●	●	S	●					
Accumulator A → CCR	TAP	06	2	1	A → CCR	<div>12</div>						●	●	●	●	●
CCR → Accumulator A	TPA	07	2	1	CCR → A	●	●	●	●	●	●					

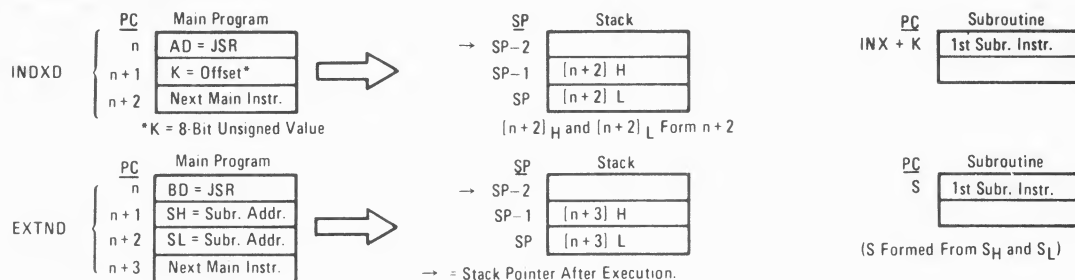
CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- | | |
|---------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| 1 (Bit V) Test: Result = 10000000? | 7 (Bit N) Test: Sign bit of most significant (MS) byte = 1? |
| 2 (Bit C) Test: Result = 00000000? | 8 (Bit V) Test: 2's complement overflow from subtraction of MS bytes? |
| 3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.) | 9 (Bit N) Test: Result less than zero? (Bit 15 = 1) |
| 4 (Bit V) Test: Operand = 10000000 prior to execution? | 10 (All) Load Condition Code Register from Stack. (See Special Operations) |
| 5 (Bit V) Test: Operand = 01111111 prior to execution? | 11 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state. |
| 6 (Bit V) Test: Set equal to result of $N \oplus C$ after shift has occurred. | 12 (All) Set according to the contents of Accumulator A. |

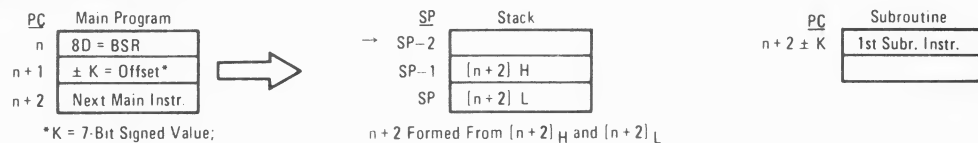


FIGURE 27 — SPECIAL OPERATIONS

JSR, JUMP TO SUBROUTINE:



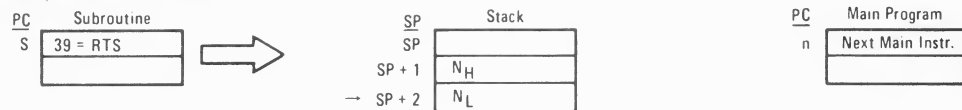
BSR, BRANCH TO SUBROUTINE:



JMP, JUMP:



RTS, RETURN FROM SUBROUTINE:



RTI, RETURN FROM INTERRUPT:

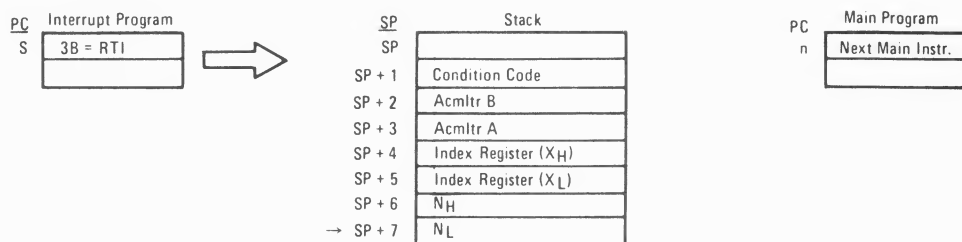


TABLE 11 — INSTRUCTION EXECUTION TIMES IN
MACHINE CYCLE

	ACCX	Immediate	Direct	Extended	Indexed	Inherent	Relative		ACCX	Immediate	Direct	Extended	Indexed	Inherent	Relative
ABA	•	•	•	•	•	2	•	INX	•	•	•	•	•	3	•
ABX	•	•	•	•	•	3	•	JMP	•	•	•	3	3	•	•
ADC	•	2	3	4	4	•	•	JSR	•	•	5	6	6	•	•
ADD	•	2	3	4	4	•	•	LDA	•	2	3	4	4	•	•
ADDD	•	4	5	6	6	•	•	LDD	•	3	4	5	5	•	•
AND	•	2	3	4	4	•	•	LDS	•	3	4	5	5	•	•
ASL	2	•	•	6	6	•	•	LDX	•	3	4	5	5	•	•
ASLD	•	•	•	•	•	3	•	LSR	2	•	•	6	6	•	•
ASR	2	•	•	6	6	•	•	LSRD	•	•	•	•	•	3	•
BCC	•	•	•	•	•	•	3	MUL	•	•	•	•	•	10	•
BCS	•	•	•	•	•	•	3	NEG	2	•	•	6	6	•	•
BEQ	•	•	•	•	•	•	3	NOP	•	•	•	•	•	2	•
BGE	•	•	•	•	•	•	3	ORA	•	2	3	4	4	•	•
BGT	•	•	•	•	•	•	3	PSH	3	•	•	•	•	•	•
BHI	•	•	•	•	•	•	3	PSHX	•	•	•	•	•	4	•
BIT	•	2	3	4	4	•	•	PUL	4	•	•	•	•	•	•
BLE	•	•	•	•	•	•	3	PULX	•	•	•	•	•	5	•
BLS	•	•	•	•	•	•	3	ROL	2	•	•	6	6	•	•
BLT	•	•	•	•	•	•	3	ROR	2	•	•	6	6	•	•
BMI	•	•	•	•	•	•	3	RTI	•	•	•	•	•	10	•
BNE	•	•	•	•	•	•	3	RTS	•	•	•	•	•	5	•
BPL	•	•	•	•	•	•	3	SBA	•	•	•	•	•	2	•
BRA	•	•	•	•	•	•	3	SBC	•	2	3	4	4	•	•
BSR	•	•	•	•	•	•	6	SEC	•	•	•	•	•	2	•
BVC	•	•	•	•	•	•	3	SEI	•	•	•	•	•	2	•
BVS	•	•	•	•	•	•	3	SEV	•	•	•	•	•	2	•
CBA	•	•	•	•	•	2	•	STA	•	•	3	4	4	•	•
CLC	•	•	•	•	•	2	•	STD	•	•	4	5	5	•	•
CLI	•	•	•	•	•	2	•	STS	•	•	4	5	5	•	•
CLR	2	•	•	6	6	•	•	STX	•	•	4	5	5	•	•
CLV	•	•	•	•	•	2	•	SUB	•	2	3	4	4	•	•
CMP	•	2	3	4	4	•	•	SUBD	•	4	5	6	6	•	•
COM	2	•	•	6	6	•	•	SWI	•	•	•	•	•	12	•
CPX	•	4	5	6	6	•	•	TAB	•	•	•	•	•	2	•
DAA	•	•	•	•	•	2	•	TAP	•	•	•	•	•	2	•
DEC	2	•	•	6	6	•	•	TBA	•	•	•	•	•	2	•
DES	•	•	•	•	•	3	•	TPA	•	•	•	•	•	2	•
DEX	•	•	•	•	•	3	•	TST	2	•	•	6	6	•	•
EOR	•	2	3	4	4	•	•	TSX	•	•	•	•	•	3	•
INC	2	•	•	6	6	•	•	TXS	•	•	•	•	•	3	•
INS	•	•	•	•	•	3	•	WAI	•	•	•	•	•	9	•



Summary of Cycle by Cycle Operation

Table 12 provides a detailed description of the information present on the Address Bus, Data Bus, and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expected results during debug of both software and hardware as the control program is executed. The information is categorized in groups according to addressing mode and number of cycles per instruction. (In general, instructions with the same addressing mode and number of cycles execute in the same manner; exceptions are indicated in the table).

TABLE 12 — CYCLE BY CYCLE OPERATION

ADDRESS MODE & INSTRUCTIONS	CYCLE	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
IMMEDIATE					
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1 2	Op Code Address Op Code Address + 1	1 1	Op Code Operand Data
LDS LDX	3	1 2 3	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte)
CPX SUBD ABDD	4	1 2 3 4	Op Code Address Op Code Address + 1 Op Code Address + 2 Address Bus FFFF	1 1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte) Low Byte of Restart Vector
DIRECT					
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1 2 3	Op Code Address Op Code Address + 1 Address of Operand	1 1 1	Op Code Address of Operand Operand Data
STA	3	1 2 3	Op Code Address Op Code Address + 1 Destination Address	1 1 0	Op Code Destination Address Data from Accumulator
LDS LDX LDD	4	1 2 3 4	Op Code Address Op Code Address + 1 Address of Operand Operand Address + 1	1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte)
STS STX STD	4	1 2 3 4	Op Code Address Op Code Address + 1 Address of Operand Address of Operand + 1	1 1 0 0	Op Code Address of Operand Register Data (High Order Byte) Register Data (Low Order Byte)
CPX SUBD ABDD	5	1 2 3 4 5	Op Code Address Op Code Address + 1 Operand Address Operand Address + 1 Address Bus FFFF	1 1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte) Low Byte of Restart Vector
JSR	5	1 2 3 4 5	Op Code Address Op Code Address + 1 Subroutine Address Stack Pointer Stack Pointer + 1	1 1 1 0 0	Op Code Irrelevant Data First Subroutine Op Code Return Address (Low Order Byte) Return Address (High Order Byte)

(continued)



TABLE 12 — CYCLE BY CYCLE OPERATION
(cont)

ADDRESS MODE & INSTRUCTIONS	CYCLES	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
INDEXED					
JMP	3	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	1	Operand Data
STA	4	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	0	Operand Data
LDS LDX LDD	5	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	1	Operand Data (High Order Byte)
		5	Index Register Plus Offset + 1	1	Operand Data (Low Order Byte)
STS STX STD	5	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	0	Operand Data (High Order Byte)
		5	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
ASL LSR ASR NEG CLR ROL COM ROR DEC TST (1) INC	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	1	Current Operand Data
		5	Address Bus FFFF	1	Low Byte of Restart Vector
		6	Index Register Plus Offset	0	New Operand Data
CPX SUBD ADDD	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register + Offset	1	Operand Data (High Order Byte)
		5	Index Register + Offset + 1	1	Operand Data (Low Order Byte)
		6	Address Bus FFFF	1	Low Byte of Restart Vector
JSR	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register + Offset	1	First Subroutine Op Code
		5	Stack Pointer	0	Return Address (Low Order Byte)
		6	Stack Pointer - 1	0	Return Address (High Order Byte)

(continued)



TABLE 12 — CYCLE BY CYCLE OPERATION
(cont)

ADDRESS MODE & INSTRUCTIONS	CYCLES	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
EXTENDED					
JMP	3	1 2 3	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Jump Address (High Order Byte) Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1 2 3 4	Op Code Address Op Code Address + 1 Op Code Address + 2 Address of Operand	1 1 1 1	Op Code Address of Operand Address of Operand (Low Order Byte) Operand Data
STA A STA B	4	1 2 3 4	Op Code Address Op Code Address + 1 Op Code Address + 2 Operand Destination Address	1 1 1 0	Op Code Destination Address (High Order Byte) Destination Address (Low Order Byte) Data from Accumulator
LDS LDX LDD	5	1 2 3	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Address of Operand (High Order Byte) Address of Operand (Low Order Byte)
STS STX STD	5	4 5 1 2 3 4 5	Address of Operand Address of Operand + 1 Op Code Address Op Code Address + 1 Op Code Address + 2 Address of Operand Address of Operand + 1	1 1 1 1 1 0 0	Operand Data (High Order Byte) Operand Data (Low Order Byte) Op Code Address of Operand (High Order Byte) Address of Operand (Low Order Byte) Operand Data (High Order Byte) Operand Data (Low Order Byte)
ASL LSR ASR NEG CLR ROL COM ROR DEC TST (1) INC	6	1 2 3 4 5 6	Op Code Address Op Code Address + 1 Op Code Address + 2 Address of Operand Address Bus FFFF Address of Operand	1 1 1 1 1 0	Op Code Address of Operand (High Order Byte) Address of Operand (Low Order Byte) Current Operand Data Low Byte of Restart Vector New Operand Data
CPX SUBD ADDD	6	1 2 3 4 5 6	Op Code Address Op Code Address + 1 Op code Address + 2 Operand Address Operand Address + 1 Address Bus FFFF	1 1 1 1 1 1	Op Code Operand Address (High Order Byte) Operand Address (Low Order Byte) Operand Data (High Order Byte) Operand Data (Low Order Byte) Low Byte of Restart Vector
JSR	6	1 2 3 4 5 6	Op Code Address Op Code Address + 1 Op Code Address + 2 Subroutine Starting Address Stack Pointer Stack Pointer - 1	1 1 1 1 0 0	Op Code Address of Subroutine (High Order Byte) Address of Subroutine (Low Order Byte) Op Code of Next Instruction Return Address (Low Order Byte) Address of Operand (High Order Byte)

(continued)

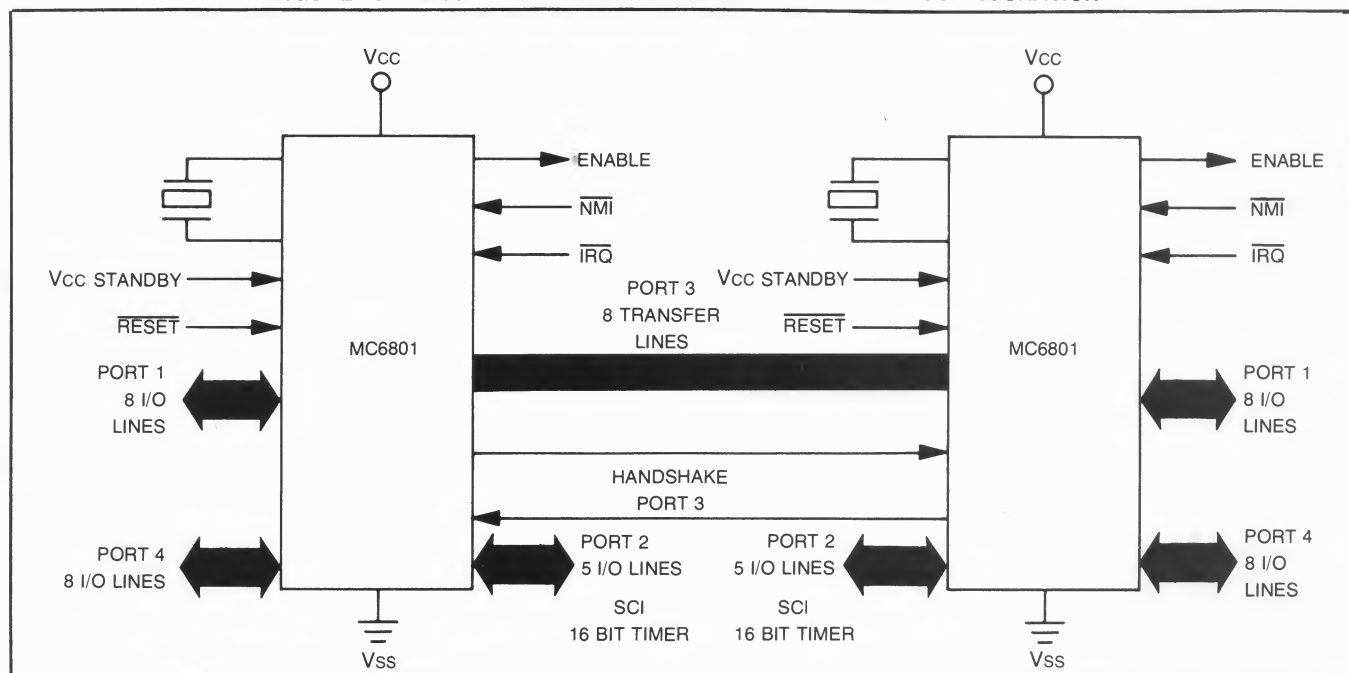
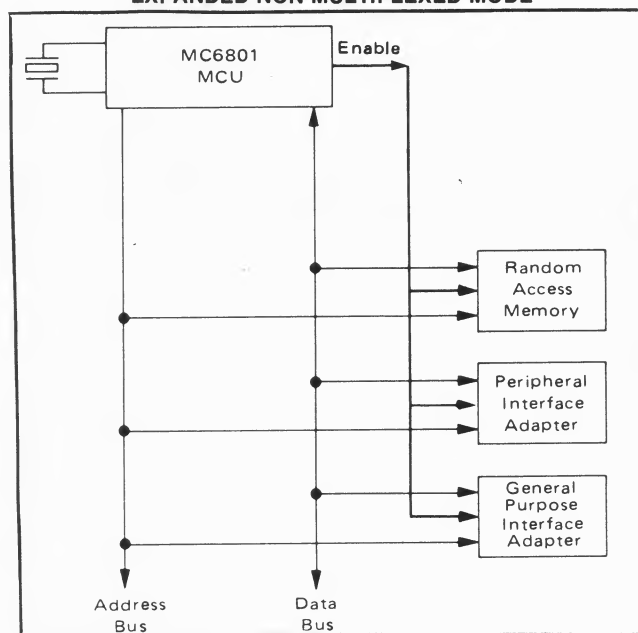
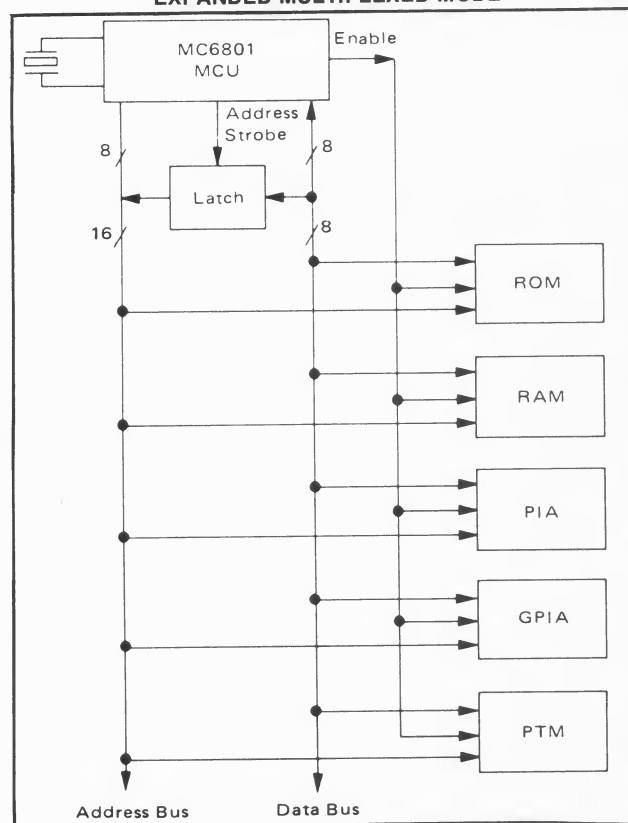


TABLE 12 — CYCLE BY CYCLE OPERATION
(cont)

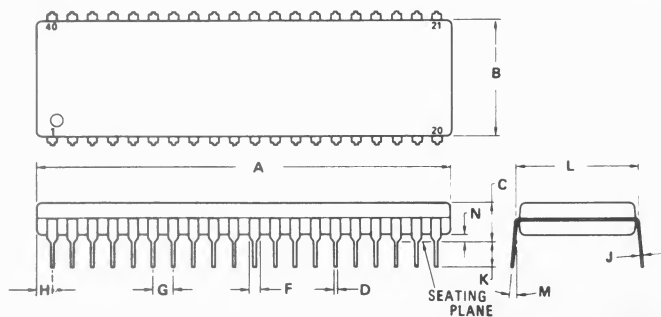
ADDRESS MODE & INSTRUCTIONS	CYCLES	CYCLES #	ADDRESS BUS	R/W LINE	DATA BUS
INHERENT					
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1 2	Op Code Address Op Code Address +1	1 1	Op Code Op Code of Next Instruction
ABX	3	1 2 3	Op Code Address Op Code Address +1 Address Bus FFFF	1 1 1	Op Code Irrelevant Data Low Byte of Restart Vector
ASLD LSRD	3	1 2 3	Op Code Address Op Code Address +1 Address Bus FFFF	1 1 1	Op Code Irrelevant Data Low Byte of Restart Vector
DES INS	3	1 2 3	Op Code Address Op Code Address +1 Previous Register Contents	1 1 1	Op Code Op Code of Next Instruction Irrelevant Data
INX DEX	3	1 2 3	Op Code Address Op Code Address +1 Address Bus FFFF	1 1 1	Op Code Op Code of Next Instruction Low Byte of Restart Vector
PSHA PSHB	3	1 2 3	Op Code Address Op Code Address +1 Stack Pointer	1 1 0	Op Code Op Code of Next Instruction Accumulator Data
ISX	3	1 2 3	Op Code Address Op Code Address +1 Stack Pointer	1 1 1	Op Code Op Code of Next Instruction Irrelevant Data
TXS	3	1 2 3	Op Code Address Op Code Address +1 Address Bus FFFF	1 1 1	Op Code Op Code of Next Instruction Low Byte of Restart Vector
PULA PULB	4	1 2 3 4	Op Code Address Op Code Address +1 Stack Pointer Stack Pointer +1	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data
PSHX	4	1 2 3 4	Op Code Address Op Code Address +1 Stack Pointer Stack Pointer -1	1 1 0 0	Op Code Irrelevant Data Index Register (Low Order Byte) Index Register (High Order Byte)
PULX	5	1 2 3 4 5	Op Code Address Op Code Address +1 Stack Pointer Stack Pointer +1 Stack Pointer +2	1 1 1 1 1	Op Code Irrelevant Data Irrelevant Data Index Register (High Order Byte) Index Register (Low Order Byte)
BCC BHT BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMT BVS	3	1 2 3	Op Code Address Op Code Address +1 Address Bus FFFF	1 1 1	Op Code Branch Offset Low Byte of Restart Vector
BSR	6	1 2 3 4 5 6	Op Code Address Op Code Address +1 Address Bus FFFF Subroutine Starting Address Stack Pointer Stack Pointer -1	1 1 1 1 0 0	Op Code Branch Offset Low Byte of Restart Vector Op Code of Next Instruction Return Address (Low Order Byte) Return Address (High Order Byte)



FIGURE 28 — MC6801 MCU SINGLE-CHIP DUAL PROCESSOR CONFIGURATION

FIGURE 29 — MC6801 MCU
EXPANDED NON-MULTIPLEXED MODEFIGURE 30 — MC6801 MCU
EXPANDED MULTIPLEXED MODE

OUTLINE DIMENSIONS

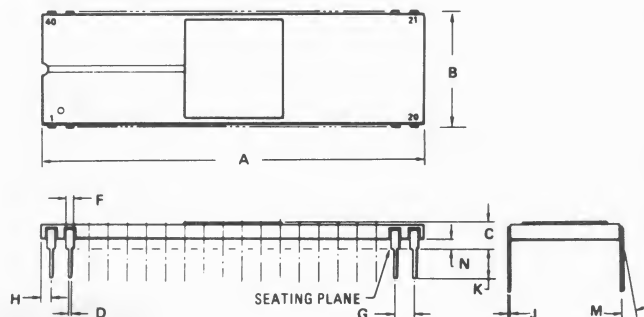


DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
M	0°	10°	0°	10°
N	0.51	1.52	0.020	0.060

L SUFFIX
CERAMIC PACKAGE
CASE 715-02

NOTE:

- LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX. MAT'L CONDITION.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.82	52.32	2.040	2.060
B	13.72	14.22	0.540	0.560
C	4.57	5.08	0.180	0.200
D	0.36	0.51	0.014	0.020
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.30	0.008	0.012
K	3.05	3.56	0.120	0.140
L	15.24 BSC		0.600 BSC	
M	0°	10°	0°	10°
N	0.51	1.02	0.020	0.040

P SUFFIX
PLASTIC PACKAGE
CASE 711-02

NOTES:

- LEADS TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA AT SEATING PLANE AT MAXIMUM MATERIAL CONDITION (DIM "D").
- DIM "L" TO CENTER OF LEADS WHEN FORMED PARALLEL.

Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Although the information in this document has been carefully reviewed for broad application, Motorola does not assume any liability arising out of the application or use of any product or circuit described herein neither does it convey any license under its patent rights nor the rights of others.





MOTOROLA Semiconductors

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

Advance Information

MICROPROCESSOR WITH CLOCK AND RAM

The MC6802 is a monolithic 8-bit microprocessor that contains all the registers and accumulators of the present MC6800 plus an internal clock oscillator and driver on the same chip. In addition, the MC6802 has 128 bytes of RAM on board located at hex addresses 0000 to 007F. The first 32 bytes of RAM, at hex addresses 0000 to 001F, may be retained in a low power mode by utilizing V_{CC} standby, thus facilitating memory retention during a power-down situation.

The MC6802 is completely software compatible with the MC6800 as well as the entire M6800 family of parts. Hence, the MC6802 is expandable to 65K words.

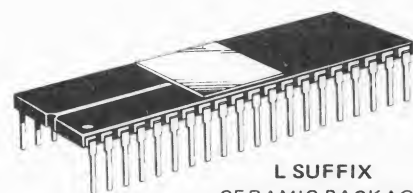
- On-Chip Clock Circuit
- 128 x 8 Bit On-Chip RAM
- 32 Bytes of RAM Are Retainable
- Software-Compatible with the MC6800
- Expandable to 65K words
- Standard TTL-Compatible Inputs and Outputs
- 8 Bit Word Size
- 16 Bit Memory Addressing
- Interrupt Capability

MC6802

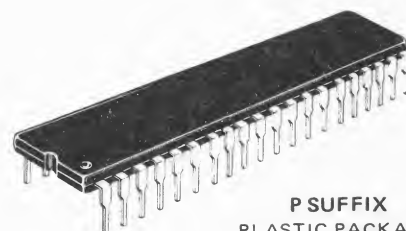
MOS

(N-CHANNEL, SILICON-GATE,
DEPLETION LOAD)

MICROPROCESSOR WITH CLOCK AND RAM



L SUFFIX
CERAMIC PACKAGE
CASE 715



P SUFFIX
PLASTIC PACKAGE
CASE 711

FIGURE 1 — TYPICAL MICROCOMPUTER

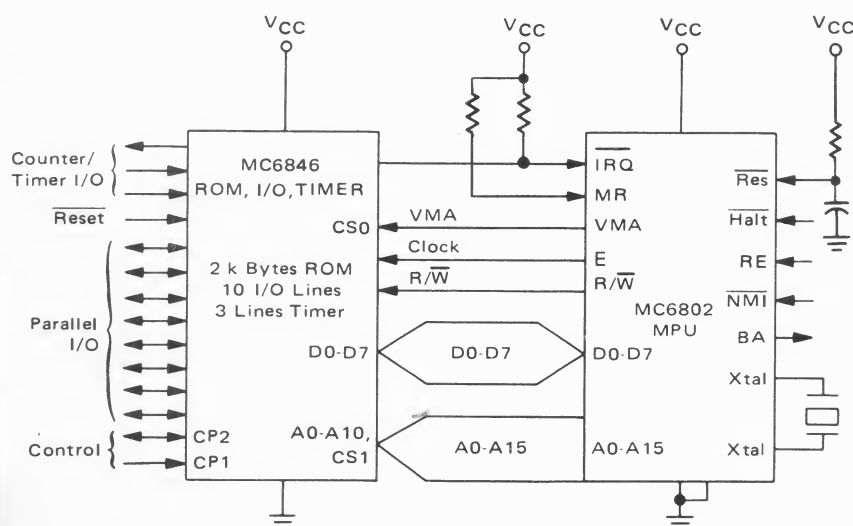


Figure 1 is a block diagram of a typical cost effective microcomputer. The MPU is the center of the microcomputer system and is shown in a minimum system interfacing with a ROM combination chip. It is not intended that this system be limited to this function but that it be expandable with other parts in the M6800 Microcomputer family.

PIN ASSIGNMENT

1	VSS	Reset	40
2	Halt	Xtal	39
3	MR	EXtal	38
4	IRQ	E	37
5	VMA	RE	36
6	NMI	VCC Standby	35
7	BA	R/W	34
8	VCC	D0	33
9	A0	D1	32
10	A1	D2	31
11	A2	D3	30
12	A3	D4	29
13	A4	D5	28
14	A5	D6	27
15	A6	D7	26
16	A7	A15	25
17	A8	A14	24
18	A9	A13	23
19	A10	A12	22
20	A11	VSS	21

B-53

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	Vdc
Input Voltage	V_{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C
Thermal Resistance	θ_{JA}	70	°C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

ELECTRICAL CHARACTERISTICS ($V_{CC} = 5.0 \text{ V} \pm 5\%$, $V_{SS} = 0$, $T_A = 0$ to 70°C unless otherwise noted.)

Characteristic		Symbol	Min	Typ	Max	Unit
Input High Voltage	Logic, EXtal Reset	V_{IH}	$V_{SS} + 2.0$ $V_{SS} + 4.0$	— —	V_{CC} V_{CC}	Vdc
Input Low Voltage	Logic, EXtal, Reset	V_{IL}	$V_{SS} - 0.3$	—	$V_{SS} + 0.8$	Vdc
Input Leakage Current ($V_{in} = 0$ to 5.25 V , $V_{CC} = \text{max}$)	Logic*	I_{in}	—	1.0	2.5	μA
Output High Voltage ($I_{Load} = -205 \mu\text{A}$, $V_{CC} = \text{min}$) ($I_{Load} = -145 \mu\text{A}$, $V_{CC} = \text{min}$) ($I_{Load} = -100 \mu\text{A}$, $V_{CC} = \text{min}$)	D0-D7 A0-A15, R/W, VMA, E BA	V_{OH}	$V_{SS} + 2.4$ $V_{SS} + 2.4$ $V_{SS} + 2.4$	— — —	— — —	Vdc
Output Low Voltage ($I_{Load} = 1.6 \text{ mA}$, $V_{CC} = \text{min}$)		V_{OL}	—	—	$V_{SS} + 0.4$	Vdc
Power Dissipation		P_D^{**}	—	0.600	1.2	W
Capacitance # ($V_{in} = 0$, $T_A = 25^\circ\text{C}$, $f = 1.0 \text{ MHz}$)	D0-D7 Logic Inputs, EXtal A0-A15, R/W, VMA	C_{in}	— —	10 6.5	12.5 10	pF
		C_{out}	—	—	12	pF
Frequency of Operation (Input Clock $\div 4$) (Crystal Frequency)		f f_{Xtal}	0.1 1.0	— —	1.0 4.0	MHz
Clock Timing						
Cycle Time		t_{cyc}	1.0	—	10	μs
Clock Pulse Width (Measured at 2.4 V)		$PW_{\phi Hs}$ $PW_{\phi L}$	450	—	4500	ns
Fall Time (Measured between $V_{SS} + 0.4 \text{ V}$ and $V_{SS} - 2.4 \text{ V}$)		t_ϕ	—	—	25	ns

*Except \overline{IRQ} and \overline{NMI} , which require $3 \text{ k}\Omega$ pullup load resistors for wire-OR capability at optimum operation. Does not include EXtal and Xtal, which are crystal inputs.

**In power-down mode, maximum power dissipation is less than 40 mW.

#Capacitances are periodically sampled rather than 100% tested.

READ/WRITE TIMING (Figures 2 through 6; Load Circuit of Figure 4.)

Characteristic	Symbol	Min	Typ	Max	Unit
Address Delay	t_{AD}	—	—	270	ns
Peripheral Read Access Time $t_{acc} = t_{ut} - (t_{AD} + t_{DSR})$	t_{acc}	—	—	530	ns
Data Setup Time (Read)	t_{DSR}	100	—	—	ns
Input Data Hold Time	t_H	10	—	—	ns
Output Data Hold Time	t_H	20	—	—	ns
Address Hold Time (Address, R/W, VMA)	t_{AH}	20	—	—	ns
Data Delay Time (Write)	t_{DDW}	—	165	225	ns
Processor Controls					
Processor Control Setup Time	t_{PCS}	200	—	—	ns
Processor Control Rise and Fall Time (Measured between 0.8 V and 2.0 V)	t_{PCr} , t_{PCf}	—	—	100	ns



FIGURE 2 — READ DATA FROM MEMORY OR PERIPHERALS

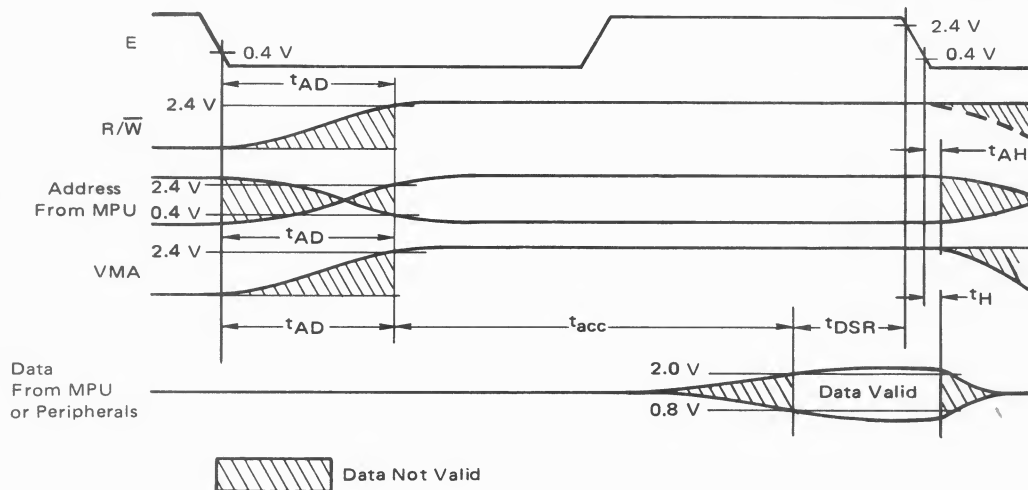


FIGURE 3 — WRITE DATA IN MEMORY OR PERIPHERALS

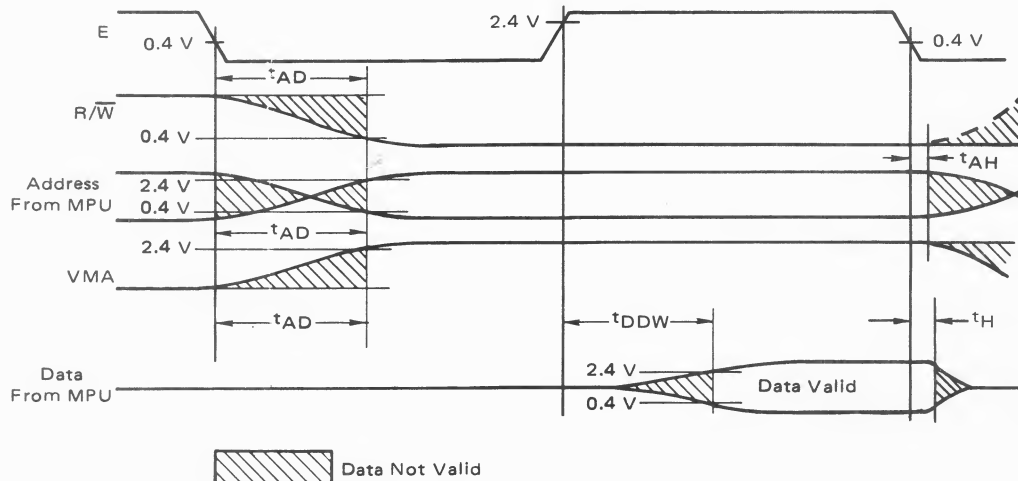


FIGURE 4 — BUS TIMING TEST LOAD

- $C = 130 \text{ pF}$ for D0-D7, E
 $= 90 \text{ pF}$ for A0-A15, R/W, and VMA
 $= 30 \text{ pF}$ for BA
 $R = 11.7 \text{ k}\Omega$ for D0-D7, E
 $= 16.5 \text{ k}\Omega$ for A0-A15, R/W, and VMA
 $= 24 \text{ k}\Omega$ for BA

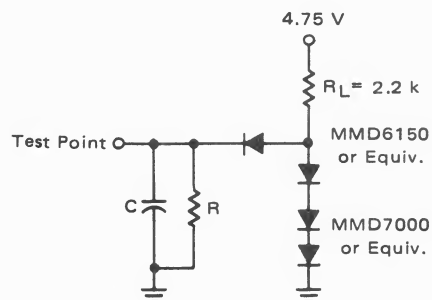


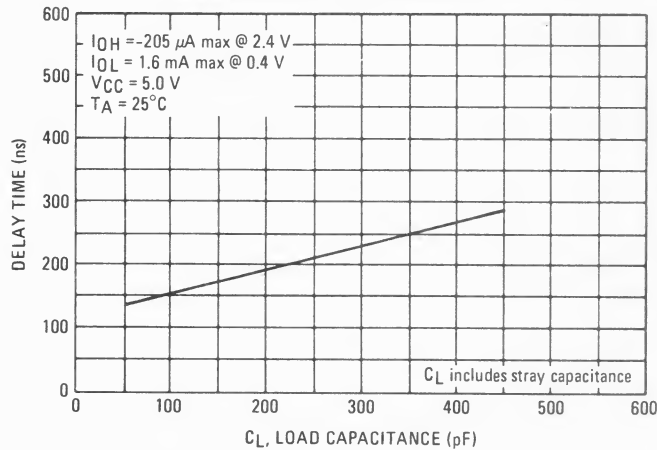
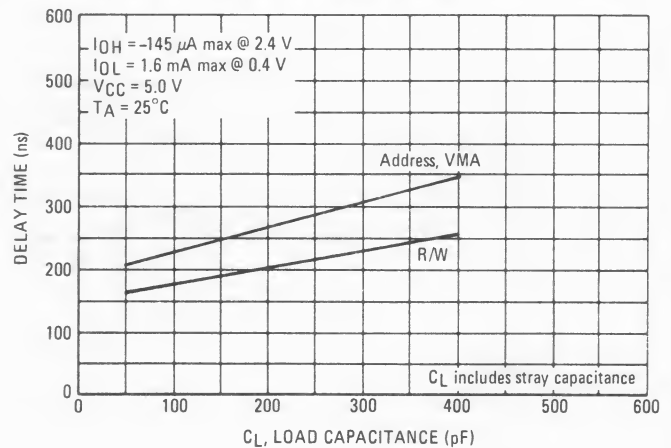
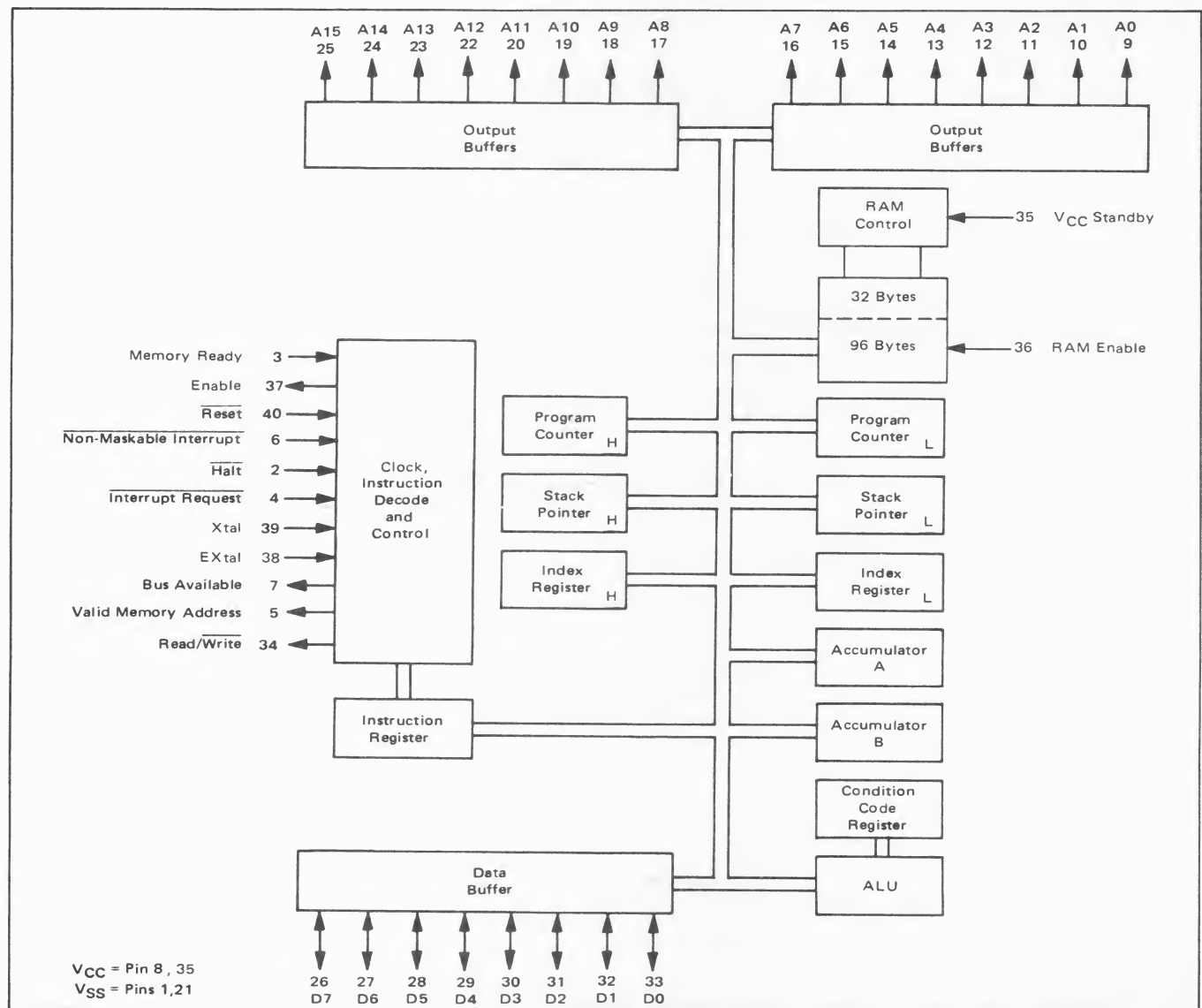
FIGURE 5 – TYPICAL DATA BUS OUTPUT DELAY
versus CAPACITIVE LOADINGFIGURE 6 – TYPICAL READ/WRITE, VMA, AND
ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING

FIGURE 7 – MC6802 EXPANDED BLOCK DIAGRAM



MPU REGISTERS

A general block diagram of the MC6802 is shown in Figure 7. As shown, the number and configuration of the registers are the same as for the MC6800. The 128 x 8 bit RAM has been added to the basic MPU. The first 32 bytes may be operated in a low power mode via a V_{CC} standby. These 32 bytes can be retained during power-up and power-down conditions via the RE signal.

The MPU has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 8).

Program Counter — The program counter is a two byte (16-bits) register that points to the current program address.

Stack Pointer — The stack pointer is a two byte register that contains the address of the next available location in an external push-down/pop-up stack. This stack is normally a random access Read/Write memory that may have any location (address) that is convenient. In those applications that require storage of information in the stack

when power is lost, the stack must be non-volatile.

Index Register — The index register is a two byte register that is used to store data or a sixteen bit memory address for the Indexed mode of memory addressing.

Accumulators — The MPU contains two 8-bit accumulators that are used to hold operands and results from an arithmetic logic unit (ALU).

Condition Code Register — The condition code register indicates the results of an Arithmetic Logic Unit operation: Negative (N), Zero (Z), Overflow (V), Carry from bit 7 (C), and half carry from bit 3 (H). These bits of the Condition Code Register are used as testable conditions for the conditional branch instructions. Bit 4 is the interrupt mask bit (I). The used bits of the Condition Code Register (b6 and b7) are ones.

Figure 9 shows the order of saving the microprocessor status within the stack.

FIGURE 8 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT

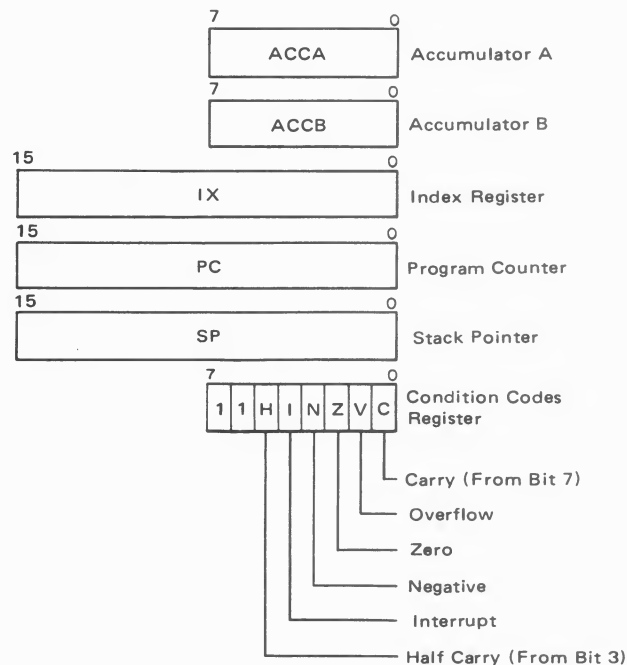
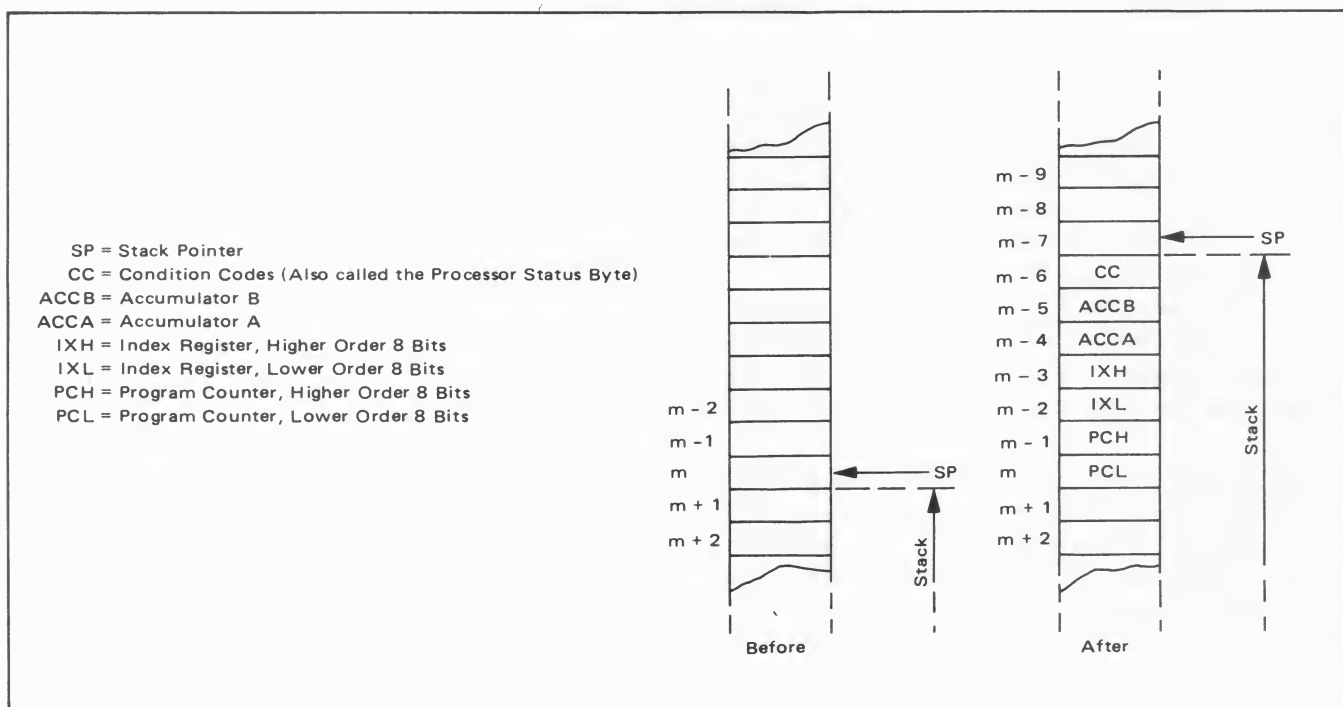


FIGURE 9 — SAVING THE STATUS OF THE MICROPROCESSOR IN THE STACK



MC6802 MPU SIGNAL DESCRIPTION

Proper operation of the MPU requires that certain control and timing signals be provided to accomplish specific functions and that other signal lines be monitored to determine the state of the processor. These control and timing signals for the MC6802 are identical to those of the MC6800 except that TSC, DBE, $\phi 1$, $\phi 2$ input, and two unused pins have been eliminated, and the following signal and timing lines have been added:

- RAM Enable (RE)
- Crystal Connections EXtal and Xtal
- Memory Ready (MR)
- V_{CC} Standby
- Enable $\phi 2$ Output (E)

The following is a summary of the MC6802 MPU signals:

Address Bus (A0-A15) — Sixteen pins are used for the address bus. The outputs are capable of driving one standard TTL load and 130 pF.

Data Bus (D0-D7) — Eight pins are used for the data bus. It is bidirectional, transferring data to and from the memory and peripheral devices. It also has three-state output buffers capable of driving one standard TTL load and 130 pF.

Halt — When this input is in the low state, all activity

in the machine will be halted. This input is level sensitive. In the halt mode, the machine will stop at the end of an instruction, Bus Available will be at a high state, Valid Memory Address will be at a low state, and all other three-state lines will be in the three-state mode. The address bus will display the address of the next instruction.

To insure single instruction operation, transition of the $\overline{\text{Halt}}$ line must not occur during the last 250 ns of E and the $\overline{\text{Halt}}$ line must go high for one Clock cycle.

Read/Write (R/\overline{W}) — This TTL compatible output signals the peripherals and memory devices whether the MPU is in a Read (high) or Write (low) state. The normal standby state of this signal is Read (high). When the processor is halted, it will be in the logical one state. This output is capable of driving one standard TTL load and 90 pF.

Valid Memory Address (VMA) — This output indicates to peripheral devices that there is a valid address on the address bus. In normal operation, this signal should be utilized for enabling peripheral interfaces such as the PIA and ACIA. This signal is not three-state. One standard TTL load and 90 pF may be directly driven by this active high signal.



Bus Available (BA) — The Bus Available signal will normally be in the low state; when activated, it will go to the high state indicating that the microprocessor has stopped and that the address bus is available. This will occur if the $\overline{\text{Halt}}$ line is in the low state or the processor is in the WAIT state as a result of the execution of a WAIT instruction. At such time, all three-state output drivers will go to their off state and other outputs to their normally inactive level. The processor is removed from the WAIT state by the occurrence of a maskable (mask bit $I = 0$) or nonmaskable interrupt. This output is capable of driving one standard TTL load and 30 pF.

Interrupt Request ($\overline{\text{IRQ}}$) — This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further interrupts may occur. At the end of the cycle, a 16-bit address will be loaded that points to a vectored address which is located in memory locations FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine

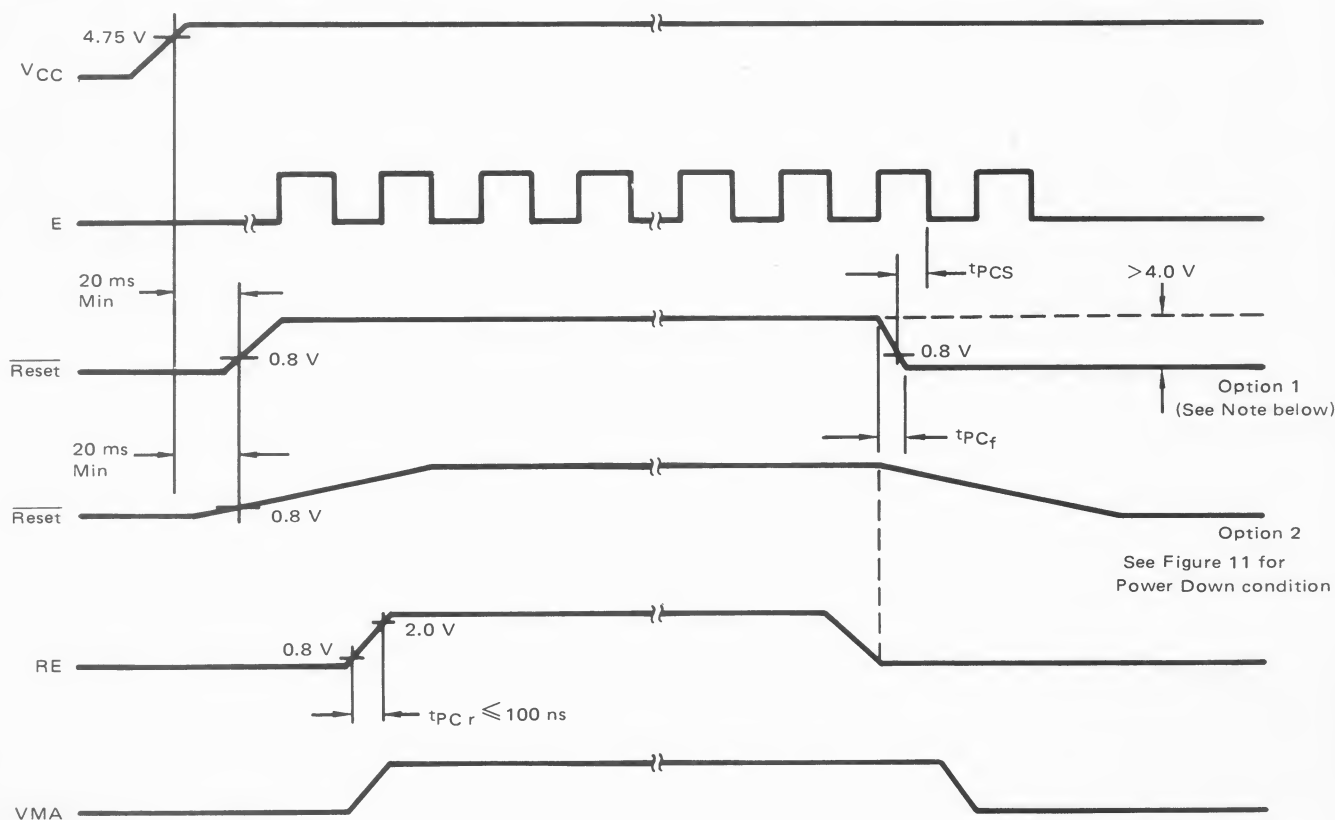
in memory.

The $\overline{\text{Halt}}$ line must be in the high state for interrupts to be serviced. Interrupts will be latched internally while $\overline{\text{Halt}}$ is low.

The $\overline{\text{IRQ}}$ has a high impedance pullup device internal to the chip; however a 3 k Ω external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

Reset — This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. When this line is low, the MPU is inactive and the information in the registers will be lost. If a high level is detected on the input, this will signal the MPU to begin the restart sequence. This will start execution of a routine to initialize the processor from its reset condition. All the higher order address lines will be forced high. For the restart, the last two (FFFE, FFFF) locations in memory will be used to load the program that is addressed by the program counter. During the restart routine, the interrupt mask bit is set and must be reset before the MPU can be interrupted by $\overline{\text{IRQ}}$. Power-up and reset timing and power-down sequences are shown in Figures 10 and 11, respectively.

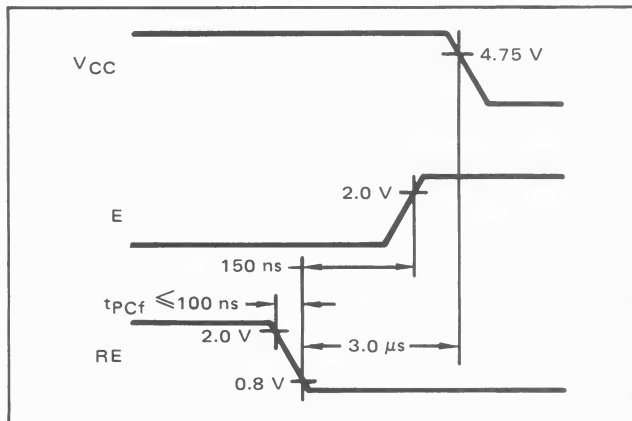
FIGURE 10 — POWER-UP AND RESET TIMING



NOTE: If option 1 is chosen, $\overline{\text{Reset}}$ and $\overline{\text{RE}}$ pins can be tied together.



FIGURE 11 — POWER-DOWN SEQUENCE



Non-Maskable Interrupt (NMI) — A low-going edge on this input requests that a non-mask-interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the NMI signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

The index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations FFFC and FFFD. An address loaded at these locations caused the MPU to branch to a non-maskable interrupt routine in memory.

NMI has a high impedance pullup resistor internal to the chip; however a 3 k Ω external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

Inputs IRQ and NMI are hardware interrupt lines that are sampled when E is high and will start the interrupt routine on a low E following the completion of an instruction.

Figure 12 is a flow chart describing the major decision paths and interrupt vectors of the microprocessor. Table 1 gives the memory map for interrupt vectors.

RAM Enable (RE) — A TTL-compatible RAM enable input controls the on-chip RAM of the MC6802. When placed in the high state, the on-chip memory is enabled to respond to the MPU controls. In the low state, RAM is disabled. This pin may also be utilized to disable reading and writing the on-chip RAM during a power-down situation. RAM enable must be low three μ s before V_{CC} goes below 4.75 V during power-down.

EXtal and Xtal — The MC6802 has an internal oscillator that may be crystal controlled. These connections are for a series resonant fundamental crystal. (AT cut.) A divide-by-four circuit has been added to the MC6802 so that a 4 MHz crystal may be used in lieu of a 1 MHz crystal for a more cost effective system. Pin 38 of the MC6802 may be driven externally by a TTL input signal if a separate clock is required. Pin 39 is to be left open in this mode.

Memory Ready (MR) — MR is a TTL compatible input control signal which allows stretching of E. When RM is high, E will be in normal operation. When MR is low, E may be stretched integral multiples of half periods, thus allowing interface to slow memories. Memory Ready timing is shown in Figure 13.

Enable (E) — This pin supplies the clock for the MPU and the rest of the system. This is a single phase, TTL compatible clock. This clock may be conditioned by a Memory Ready Signal. This is equivalent to $\phi 2$ on the MC6800.

V_{CC} Standby — This pin supplies the dc voltage to the first 32 bytes of RAM as well as the RAM Enable (RE) control logic. Thus retention of data in this portion of the RAM on a power-up, power-down, or standby condition is guaranteed. Maximum current drain at 5.25 V is 8 mA.

TABLE 1 — MEMORY MAP FOR INTERRUPT VECTORS

Vector		Description
MS	LS	
FFFE	FFFF	Restart
FFFC	FFFD	Non-maskable Interrupt
FFFA	FFFB	Software Interrupt
FFF8	FFF9	Interrupt Request



FIGURE 12 – MPU FLOW CHART

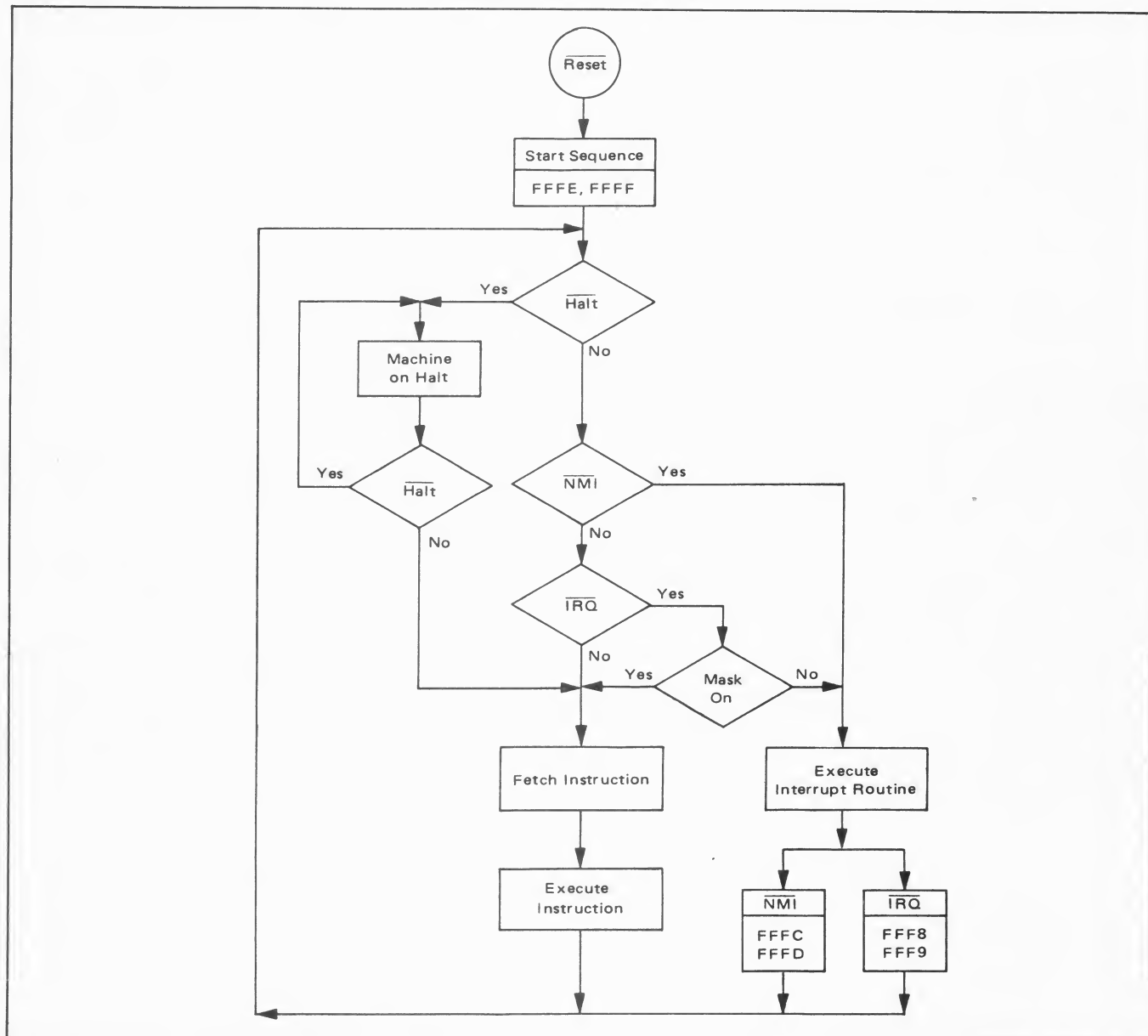
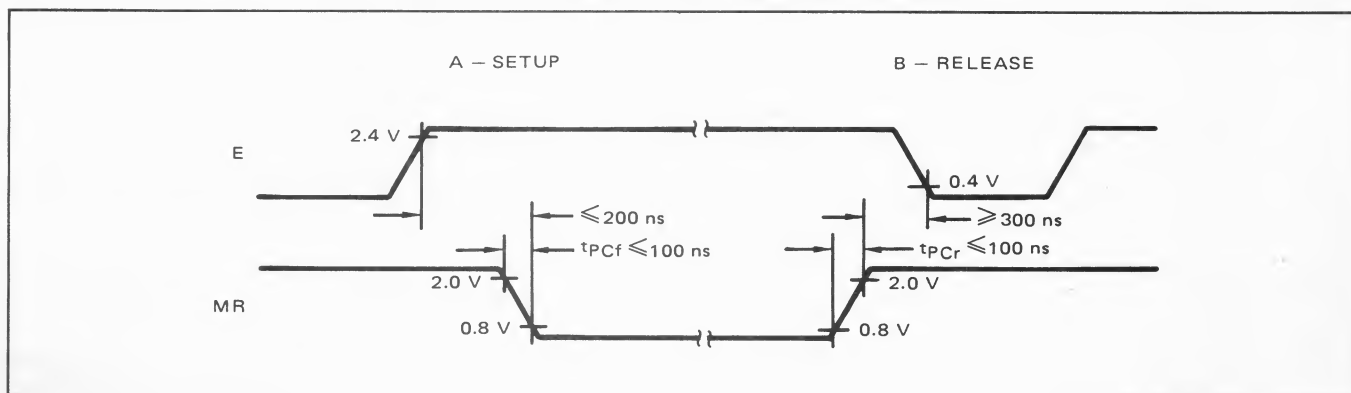


FIGURE 13 – MEMORY READY CONTROL FUNCTION



MPU INSTRUCTION SET

The MC6802 has a set of 72 different instructions. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, conditional or unconditional branch, interrupt and stack manipulation instructions (Tables 2 thru 6). This instruction set is the same as that for the MC6800.

MPU ADDRESSING MODES

The MC6802 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 7 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 1 MHz, these times would be microseconds.

Accumulator (ACCX) Addressing — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

Immediate Addressing — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MPU addresses this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

Direct Addressing — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

Extended Addressing — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

Indexed Addressing — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

Implied Addressing — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

Relative Addressing — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.

TABLE 2 — MICROPROCESSOR INSTRUCTION SET — ALPHABETIC SEQUENCE

ABA	Add Accumulators	CLR	Clear	PUL	Pull Data
ADC	Add with Carry	CLV	Clear Overflow	ROL	Rotate Left
ADD	Add	CMP	Compare	ROR	Rotate Right
AND	Logical And	COM	Complement	RTI	Return from Interrupt
ASL	Arithmetic Shift Left	CPX	Compare Index Register	RTS	Return from Subroutine
ASR	Arithmetic Shift Right	DAA	Decimal Adjust	SBA	Subtract Accumulators
BCC	Branch if Carry Clear	DEC	Decrement	SBC	Subtract with Carry
BCS	Branch if Carry Set	DES	Decrement Stack Pointer	SEC	Set Carry
BEQ	Branch if Equal to Zero	DEX	Decrement Index Register	SEI	Set Interrupt Mask
BGE	Branch if Greater or Equal Zero	EOR	Exclusive OR	SEV	Set Overflow
BGT	Branch if Greater than Zero	INC	Increment	STA	Store Accumulator
BHI	Branch if Higher	INS	Increment Stack Pointer	STS	Store Stack Register
BIT	Bit Test	INX	Increment Index Register	STX	Store Index Register
BLE	Branch if Less or Equal	JMP	Jump	SUB	Subtract
BLS	Branch if Lower or Same	JSR	Jump to Subroutine	SWI	Software Interrupt
BLT	Branch if Less than Zero	LDA	Load Accumulator	TAB	Transfer Accumulators
BMI	Branch if Minus	LDS	Load Stack Pointer	TAP	Transfer Accumulators to Condition Code Reg.
BNE	Branch if Not Equal to Zero	LDX	Load Index Register	TBA	Transfer Accumulators
BPL	Branch if Plus	LSR	Logical Shift Right	TPA	Transfer Condition Code Reg. to Accumulator
BRA	Branch Always	NEG	Negate	TST	Test
BSR	Branch to Subroutine	NOP	No Operation	TSX	Transfer Stack Pointer to Index Register
BVC	Branch if Overflow Clear	ORA	Inclusive OR Accumulator	TXS	Transfer Index Register to Stack Pointer
BVS	Branch if Overflow Set	PSH	Push Data	WAI	Wait for Interrupt
CBA	Compare Accumulators				
CLC	Clear Carry				
CLI	Clear Interrupt Mask				



TABLE 3 — ACCUMULATOR AND MEMORY INSTRUCTIONS

		ADDRESSING MODES										BOOLEAN/ARITHMETIC OPERATION				COND. CODE REG.					
		IMMED		DIRECT		INDEX		EXTND		IMPLIED		(All register labels refer to contents)				5	4	3	2	1	0
OPERATIONS	MNEMONIC	OP	~ =	OP	~ =	OP	~ =	OP	~ =	OP	~ =					H	I	N	Z	V	C
Add	ADDA	38	2 2	98	3 2	A8	5 2	B8	4 3			A + M → A	•	•	•	•	•	•	•	•	
	ADDB	CB	2 2	D8	3 2	E8	5 2	F8	4 3			B + M → B	•	•	•	•	•	•	•	•	
Add Acmltrs	ABA									1B	2 1	A + B → A	•	•	•	•	•	•	•	•	
Add with Carry	ADCA	89	2 2	99	3 2	A9	5 2	B9	4 3			A + M + C → A	•	•	•	•	•	•	•	•	
	ADCB	C9	2 2	D9	3 2	E9	5 2	F9	4 3			B + M + C → B	•	•	•	•	•	•	•	•	
And	ANDA	84	2 2	94	3 2	A4	5 2	B4	4 3			A · M → A	•	•	•	•	•	R	•	•	
	ANDB	C4	2 2	D4	3 2	E4	5 2	F4	4 3			B · M → B	•	•	•	•	•	R	•	•	
Bit Test	BITA	85	2 2	95	3 2	A5	5 2	B5	4 3			A · M	•	•	•	•	•	R	•	•	
	BITB	C5	2 2	D5	3 2	E5	5 2	F5	4 3			B · M	•	•	•	•	•	R	•	•	
Clear	CLR					6F	7 2	7F	6 3			00 → M	•	•	•	R	S	R	R	•	
	CLRA									4F	2 1	00 → A	•	•	•	R	S	R	R	•	
	CLRB									5F	2 1	00 → B	•	•	•	R	S	R	R	•	
Compare	CMPA	81	2 2	91	3 2	A1	5 2	B1	4 3			A - M	•	•	•	•	•	•	•	•	
	CMPB	C1	2 2	D1	3 2	E1	5 2	F1	4 3			B - M	•	•	•	•	•	•	•	•	
Compare Acmltrs	CBA									11	2 1	A - B	•	•	•	•	•	•	•	•	
Complement, 1's	COM					63	7 2	73	6 3			M → M	•	•	•	•	•	R	S	•	
	COMA									43	2 1	A̅ → A	•	•	•	•	•	R	S	•	
	COMB									53	2 1	B̅ → B	•	•	•	•	•	R	S	•	
Complement, 2's (Negate)	NEG					60	7 2	70	6 3			00 - M → M	•	•	•	•	•	①	②	•	
	NEGA									40	2 1	00 - A → A	•	•	•	•	•	①	②	•	
	NEGB									50	2 1	00 - B → B	•	•	•	•	•	①	②	•	
Decimal Adjust, A	DAA									19	2 1	Converts Binary Add. of BCD Characters into BCD Format	•	•	•	•	•	•	③	•	
Decrement	DEC					6A	7 2	7A	6 3			M - 1 → M	•	•	•	•	•	4	•	•	
	DECA									4A	2 1	A - 1 → A	•	•	•	•	•	4	•	•	
	DECB									5A	2 1	B - 1 → B	•	•	•	•	•	4	•	•	
Exclusive OR	EORA	88	2 2	98	3 2	A8	5 2	B8	4 3			A ⊕ M → A	•	•	•	•	•	R	•	•	
	EORB	C8	2 2	D8	3 2	E8	5 2	F8	4 3			B ⊕ M → B	•	•	•	•	•	R	•	•	
Increment	INC					6C	7 2	7C	6 3			M + 1 → M	•	•	•	•	•	⑤	•	•	
	INCA									4C	2 1	A + 1 → A	•	•	•	•	•	⑤	•	•	
	INCB									5C	2 1	B + 1 → B	•	•	•	•	•	⑤	•	•	
Load Acmltr	LDAA	86	2 2	96	3 2	A6	5 2	B6	4 3			M → A	•	•	•	•	•	R	•	•	
	LDAB	C6	2 2	D6	3 2	E6	5 2	F6	4 3			M → B	•	•	•	•	•	R	•	•	
Or, Inclusive	ORAA	8A	2 2	9A	3 2	AA	5 2	BA	4 3			A + M → A	•	•	•	•	•	R	•	•	
	ORAB	CA	2 2	DA	3 2	EA	5 2	FA	4 3			B + M → B	•	•	•	•	•	R	•	•	
Push Data	PSHA									36	4 1	A → Msp, SP - 1 → SP	•	•	•	•	•	R	•	•	
	PSHB									37	4 1	B → Msp, SP - 1 → SP	•	•	•	•	•	R	•	•	
Pull Data	PULA									32	4 1	SP + 1 → SP, Msp → A	•	•	•	•	•	•	•	•	
	PULB									33	4 1	SP + 1 → SP, Msp → B	•	•	•	•	•	•	•	•	
Rotate Left	ROL					69	7 2	79	6 3			M	•	•	•	•	•	⑥	•	•	
	ROLA									49	2 1	A	•	•	•	•	•	⑥	•	•	
	ROLB									59	2 1	B	•	•	•	•	•	⑥	•	•	
Rotate Right	ROR					66	7 2	76	6 3			M	•	•	•	•	•	⑥	•	•	
	RORA									46	2 1	A	•	•	•	•	•	⑥	•	•	
	RORB									56	2 1	B	•	•	•	•	•	⑥	•	•	
Shift Left, Arithmetic	ASL					68	7 2	78	6 3			M	•	•	•	•	•	⑥	•	•	
	ASLA									48	2 1	A	•	•	•	•	•	⑥	•	•	
	ASLB									58	2 1	B	•	•	•	•	•	⑥	•	•	
Shift Right, Arithmetic	ASR					67	7 2	77	6 3			M	•	•	•	•	•	⑥	•	•	
	ASRA									47	2 1	A	•	•	•	•	•	⑥	•	•	
	ASRB									57	2 1	B	•	•	•	•	•	⑥	•	•	
Shift Right, Logic	LSR					64	7 2	74	6 3			M	•	•	•	•	•	⑥	•	•	
	LSRA									44	2 1	A	•	•	•	•	•	⑥	•	•	
	LSRB									54	2 1	B	•	•	•	•	•	⑥	•	•	
Store Acmltr.	STAA			97	4 2	A7	6 2	B7	5 3			A → M	•	•	•	•	•	R	•	•	
	STAB			D7	4 2	E7	6 2	F7	5 3			B → M	•	•	•	•	•	R	•	•	
Subtract	SUBA	80	2 2	90	3 2	A0	5 2	B0	4 3			A - M → A	•	•	•	•	•	•	•	•	
	SUBB	C0	2 2	D0	3 2	E0	5 2	F0	4 3			B - M → B	•	•	•	•	•	•	•	•	
Subtract Acmltrs.	SBA									10	2 1	A - B → A	•	•	•	•	•	•	•	•	
Subtr. with Carry	SBCA	82	2 2	92	3 2	A2	5 2	B2	4 3			A - M - C → A	•	•	•	•	•	•	•	•	
	SBCB	C2	2 2	D2	3 2	E2	5 2	F2	4 3			B - M - C → B	•	•	•	•	•	•	•	•	
Transfer Acmltrs	TAB									16	2 1	A → B	•	•	•	•	•	•	R	•	
	TBA									17	2 1	B → A	•	•	•	•	•	•	R	•	
Test, Zero or Minus	TST					6D	7 2	7D	6 3			M - 00	•	•	•	•	•	•	R	R	
	TSTA									4D	2 1	A - 00	•	•	•	•	•	•	R	R	
	TSTB									5D	2 1	B - 00	•	•	•	•	•	•	R	R	

LEGEND:

OP Operation Code (Hexadecimal);
 ~ Number of MPU Cycles;
 = Number of Program Bytes;
 + Arithmetic Plus;
 - Arithmetic Minus;
 • Boolean AND;
 Msp Contents of memory location pointed to be Stack Pointer;

+ Boolean Inclusive OR;
 ⊖ Boolean Exclusive OR;
 M Complement of M;
 → Transfer Into;
 0 Bit = Zero;
 00 Byte = Zero;

CONDITION CODE SYMBOLS:

H Half-carry from bit 3;
 I Interrupt mask;
 N Negative (sign bit);
 Z Zero (byte);
 V Overflow, 2's complement;
 C Carry from bit 7;
 R Reset Always;
 S Set Always;
 ! Test and set if true, cleared otherwise;
 • Not Affected

Note — Accumulator addressing mode instructions are included in the column for IMPLIED addressing



TABLE 4 — INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

																	COND. CODE REG.						
POINTER OPERATIONS	MNEMONIC	IMMED			DIRECT			INDEX			EXTND			IMPLIED			BOOLEAN/ARITHMETIC OPERATION	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	8C	5	3				$X_H - M, X_L - (M + 1)$	●	●	⑦	↑	⑧	●
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	●	●	●	↑	●	●
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	●	●	●	●	●	●
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	●	●	●	↑	●	●
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	●	●	●	●	●	●
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	●	●	⑨	↑	R	●
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	●	●	⑨	↑	R	●
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	●	●	⑨	↑	R	●
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	●	●	⑨	↑	R	●
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	●	●	●	●	●	●
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	●	●	●	●	●	●

TABLE 5 — JUMP AND BRANCH INSTRUCTIONS

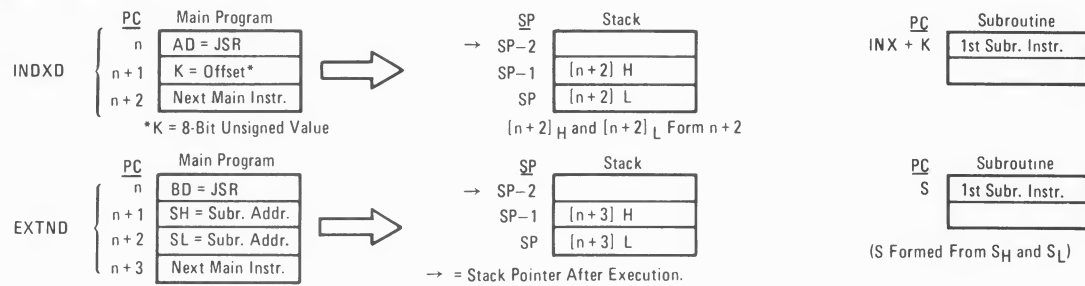
														COND. CODE REG.							
		RELATIVE			INDEX			EXTND			IMPLIED										
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	BRANCH TEST	5	4	3	2	1	0	
																H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•	
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•	
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•	
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•	
Branch If ≥ Zero	BGE	2C	4	2										N ⊕ V = 0	•	•	•	•	•	•	
Branch If > Zero	BGT	2E	4	2										Z + (N ⊕ V) = 0	•	•	•	•	•	•	
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•	
Branch If ≤ Zero	BLE	2F	4	2										Z + (N ⊕ V) = 1	•	•	•	•	•	•	
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•	
Branch If < Zero	BLT	2D	4	2										N ⊕ V = 1	•	•	•	•	•	•	
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•	
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•	
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•	
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•	
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•	
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•	
Jump	JMP				6E	4	2	7E	3	3				} See Special Operations	•	•	•	•	•	•	
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•	
Return From Interrupt	RTI										3B	10	1		•	•	•	•	•	•	•
Return From Subroutine	RTS										39	5	1	} See Special Operations	•	•	•	•	•	•	
Software Interrupt	SWI										3F	12	1		•	•	•	•	•	•	•
Wait for Interrupt*	WAI										3E	9	1		•	11	•	•	•	•	•

*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

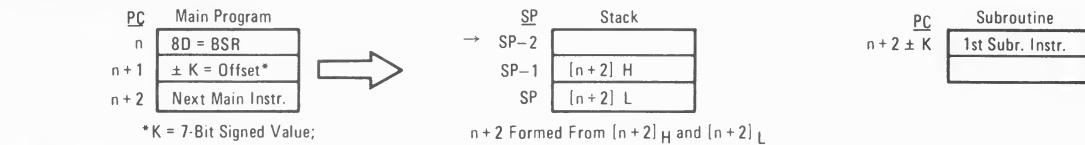


SPECIAL OPERATIONS

JSR, JUMP TO SUBROUTINE:



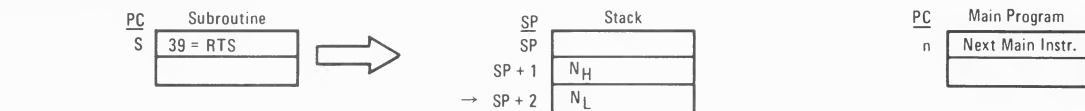
BSR, BRANCH TO SUBROUTINE:



JMP, JUMP:



RTS, RETURN FROM SUBROUTINE:



RTI, RETURN FROM INTERRUPT:

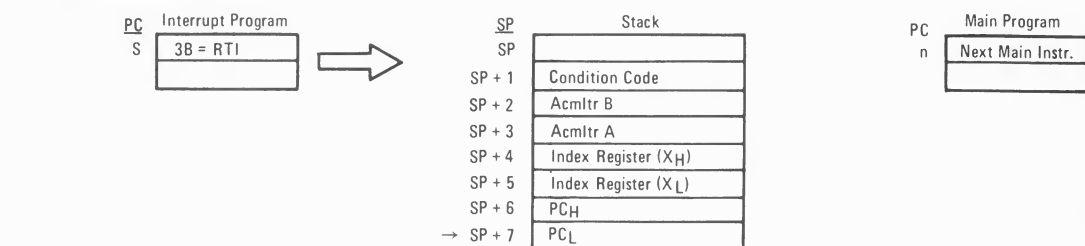


TABLE 6 – CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

						COND. CODE REG.						
OPERATIONS	MNEMONIC	IMPLIED			BOOLEAN OPERATION	5	4	3	2	1	0	
		OP	~	=		H	I	N	Z	V	C	
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	●	●	●	●	●	R	
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	●	R	●	●	●	●	
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	●	●	●	●	R	●	
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	●	●	●	●	●	S	
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	●	S	●	●	●	●	
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	●	●	●	●	S	●	
Accmltr A \rightarrow CCR	TAP	06	2	1	A \rightarrow CCR	(12)						●
CCR \rightarrow Accmltr A	TPA	07	2	1	CCR \rightarrow A							●

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- | | |
|------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| 1 (Bit V) Test: Result = 10000000? | 7 (Bit N) Test: Sign bit of most significant (MS) byte = i? |
| 2 (Bit C) Test: Result ≠ 00000000? | 8 (Bit V) Test: 2's complement overflow from subtraction of MS bytes? |
| 3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine?
(Not cleared if previously set.) | 9 (Bit N) Test: Result less than zero? (Bit 15 = 1) |
| 4 (Bit V) Test: Operand = 10000000 prior to execution? | 10 (All) Load Condition Code Register from Stack. (See Special Operations) |
| 5 (Bit V) Test: Operand = 01111111 prior to execution? | 11 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state. |
| 6 (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred. | 12 (All) Set according to the contents of Accumulator A. |



TABLE 7 — INSTRUCTION ADDRESSING MODES AND ASSOCIATED EXECUTION TIMES
(Times in Machine Cycles)

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied	Relative		(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied
ABA		•	•	•	•	•	2	•	INC	2	•	•	•	6	7	•
ADC	x	•	2	3	4	5	•	•	INS	•	•	•	•	•	•	4
ADD	x	•	2	3	4	5	•	•	INX	•	•	•	•	•	•	4
AND	x	•	2	3	4	5	•	•	JMP	•	•	•	3	4	•	•
ASL	2	•	•	6	7	•	•	•	JSR	•	•	•	9	8	•	•
ASR	2	•	•	6	7	•	•	•	LDA	x	2	3	4	5	•	•
BCC	•	•	•	•	•	•	•	4	LDS	•	3	4	5	6	•	•
BCS	•	•	•	•	•	•	•	4	LDX	•	3	4	5	6	•	•
BEA	•	•	•	•	•	•	•	4	LSR	2	•	•	6	7	•	•
BGE	•	•	•	•	•	•	•	4	NEG	2	•	•	6	7	•	•
BGT	•	•	•	•	•	•	•	4	NOP	•	•	•	•	•	•	2
BHI	•	•	•	•	•	•	•	4	ORA	x	2	3	4	5	•	•
BIT	x	•	2	3	4	5	•	•	PSH	•	•	•	•	•	•	4
BLE	•	•	•	•	•	•	•	4	PUL	•	•	•	•	•	•	4
BLS	•	•	•	•	•	•	•	4	ROL	2	•	•	6	7	•	•
BLT	•	•	•	•	•	•	•	4	ROR	2	•	•	6	7	•	•
BMI	•	•	•	•	•	•	•	4	RTI	•	•	•	•	•	10	•
BNE	•	•	•	•	•	•	•	4	RTS	•	•	•	•	•	5	•
BPL	•	•	•	•	•	•	•	4	SBA	•	•	•	•	•	•	2
BRA	•	•	•	•	•	•	•	4	SBC	x	2	3	4	5	•	•
BSR	•	•	•	•	•	•	•	8	SEC	•	•	•	•	•	•	2
BVC	•	•	•	•	•	•	•	4	SEI	•	•	•	•	•	•	2
BVS	•	•	•	•	•	•	•	4	SEV	•	•	•	•	•	•	2
CBA	•	•	•	•	•	2	•	•	STA	x	•	4	5	6	•	•
CLC	•	•	•	•	•	2	•	•	STS	•	•	5	6	7	•	•
CLI	•	•	•	•	•	2	•	•	STX	•	•	5	6	7	•	•
CLR	2	•	•	6	7	•	•	•	SUB	x	•	2	3	4	5	•
CLV	•	•	•	•	•	2	•	•	SWI	•	•	•	•	•	12	•
CMP	x	•	2	3	4	5	•	•	TAB	•	•	•	•	•	•	2
COM	2	•	•	6	7	•	•	•	TAP	•	•	•	•	•	•	2
CPX	•	3	4	5	6	•	•	•	TBA	•	•	•	•	•	•	2
DAA	•	•	•	•	•	2	•	•	TPA	•	•	•	•	•	•	2
DEC	2	•	•	6	7	•	•	•	TST	2	•	•	6	7	•	•
DES	•	•	•	•	•	4	•	•	TSX	•	•	•	•	•	•	4
DEX	•	•	•	•	•	4	•	•	TSX	•	•	•	•	•	•	4
EOR	x	•	2	3	4	5	•	•	WAI	•	•	•	•	•	•	9

NOTE: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.



SUMMARY OF CYCLE BY CYCLE OPERATION

Table 8 provides a detailed description of the information present on the Address Bus, Data Bus, Valid Memory Address line (VMA), and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expected results during debug of both software and hard-

ware as the control program is executed. The information is categorized in groups according to Addressing Mode and Number of Cycles per instruction. (In general, instructions with the same Addressing Mode and Number of Cycles execute in the same manner; exceptions are indicated in the table.)

TABLE 8 – OPERATION SUMMARY

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
IMMEDIATE						
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1 2	1 1	Op Code Address Op Code Address + 1	1 1	Op Code Operand Data
CPX LDS LDX	3	1 2 3	1 1 1	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte)
DIRECT						
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1 2 3	1 1 1	Op Code Address Op Code Address + 1 Address of Operand	1 1 1	Op Code Address of Operand Operand Data
CPX LDS LDX	4	1 2 3 4	1 1 1 1	Op Code Address Op Code Address + 1 Address of Operand Operand Address + 1	1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte)
STA	4	1 2 3 4	1 1 0 1	Op Code Address Op Code Address + 1 Destination Address Destination Address	1 1 1 0	Op Code Destination Address Irrelevant Data (Note 1) Data from Accumulator
STS STX	5	1 2 3 4 5	1 1 0 1 1	Op Code Address Op Code Address + 1 Address of Operand Address of Operand Address of Operand + 1	1 1 1 0 0	Op Code Address of Operand Irrelevant Data (Note 1) Register Data (High Order Byte) Register Data (Low Order Byte)
INDEXED						
JMP	4	1 2 3 4	1 1 0 0	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry)	1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	5	1 2 3 4 5	1 1 0 0 1	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry) Index Register Plus Offset	1 1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1) Operand Data
CPX LDS LDX	6	1 2 3 4 5 6	1 1 0 0 1 1	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry) Index Register Plus Offset Index Register Plus Offset + 1	1 1 1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1) Operand Data (High Order Byte) Operand Data (Low Order Byte)



TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INDEXED (Continued)						
STA	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	1	Index Register Plus Offset	1	Current Operand Data
		6	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		7	1/0 (Note 3)	Index Register Plus Offset	0	New Operand Data (Note 3)
STS STX	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data (High Order Byte)
		7	1	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
JSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer — 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer — 2	1	Irrelevant Data (Note 1)
		7	0	Index Register	1	Irrelevant Data (Note 1)
		8	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
EXTENDED						
JMP	3	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Jump Address (High Order Byte)
		3	1	Op Code Address + 2	1	Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data
CPX LDS LDX	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data (High Order Byte)
		5	1	Address of Operand + 1	1	Operand Data (Low Order Byte)
STA A STA B	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Destination Address (High Order Byte)
		3	1	Op Code Address + 2	1	Destination Address (Low Order Byte)
		4	0	Operand Destination Address	1	Irrelevant Data (Note 1)
		5	1	Operand Destination Address	0	Data from Accumulator
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Current Operand Data
		5	0	Address of Operand	1	Irrelevant Data (Note 1)
		6	1/0 (Note 3)	Address of Operand	0	New Operand Data (Note 3)

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus	
EXTENDED (Continued)							
STS STX	6	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)	
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)	
		4	0	Address of Operand	1	Irrelevant Data (Note 1)	
		5	1	Address of Operand	0	Operand Data (High Order Byte)	
		6	1	Address of Operand + 1	0	Operand Data (Low Order Byte)	
JSR	9	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Address of Subroutine (High Order Byte)	
		3	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)	
		4	1	Subroutine Starting Address	1	Op Code of Next Instruction	
		5	1	Stack Pointer	0	Return Address (Low Order Byte)	
		6	1	Stack Pointer — 1	0	Return Address (High Order Byte)	
		7	0	Stack Pointer — 2	1	Irrelevant Data (Note 1)	
		8	0	Op Code Address + 2	1	Irrelevant Data (Note 1)	
		9	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)	
INHERENT							
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Op Code of Next Instruction	
	DES DEX INS INX	4	1	1	Op Code Address	1	Op Code
			2	1	Op Code Address + 1	1	Op Code of Next Instruction
			3	0	Previous Register Contents	1	Irrelevant Data (Note 1)
			4	0	New Register Contents	1	Irrelevant Data (Note 1)
	PSH	4	1	1	Op Code Address	1	Op Code
			2	1	Op Code Address + 1	1	Op Code of Next Instruction
3			1	Stack Pointer	0	Accumulator Data	
4			0	Stack Pointer — 1	1	Accumulator Data	
PUL	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Op Code of Next Instruction	
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)	
		4	1	Stack Pointer + 1	1	Operand Data from Stack	
TSX	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Op Code of Next Instruction	
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)	
		4	0	New Index Register	1	Irrelevant Data (Note 1)	
TXS	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Op Code of Next Instruction	
		3	0	Index Register	1	Irrelevant Data	
		4	0	New Stack Pointer	1	Irrelevant Data	
RTS	5	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)	
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)	
		4	1	Stack Pointer + 1	1	Address of Next Instruction (High Order Byte)	
		5	1	Stack Pointer + 2	1	Address of Next Instruction (Low Order Byte)	



TABLE 8 – OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus	
INHERENT (Continued)							
WAI	9	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Op Code of Next Instruction	
		3	1	Stack Pointer	0	Return Address (Low Order Byte)	
		4	1	Stack Pointer – 1	0	Return Address (High Order Byte)	
		5	1	Stack Pointer – 2	0	Index Register (Low Order Byte)	
		6	1	Stack Pointer – 3	0	Index Register (High Order Byte)	
		7	1	Stack Pointer – 4	0	Contents of Accumulator A	
		8	1	Stack Pointer – 5	0	Contents of Accumulator B	
		9	1	Stack Pointer – 6 (Note 4)	1	Contents of Cond. Code Register	
RTI	10	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)	
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)	
		4	1	Stack Pointer + 1	1	Contents of Cond. Code Register from Stack	
		5	1	Stack Pointer + 2	1	Contents of Accumulator B from Stack	
		6	1	Stack Pointer + 3	1	Contents of Accumulator A from Stack	
		7	1	Stack Pointer + 4	1	Index Register from Stack (High Order Byte)	
		8	1	Stack Pointer + 5	1	Index Register from Stack (Low Order Byte)	
		9	1	Stack Pointer + 6	1	Next Instruction Address from Stack (High Order Byte)	
			10	1	Stack Pointer + 7	1	Next Instruction Address from Stack (Low Order Byte)
SWI	12	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 1)	
		3	1	Stack Pointer	0	Return Address (Low Order Byte)	
		4	1	Stack Pointer – 1	0	Return Address (High Order Byte)	
		5	1	Stack Pointer – 2	0	Index Register (Low Order Byte)	
		6	1	Stack Pointer – 3	0	Index Register (High Order Byte)	
		7	1	Stack Pointer – 4	0	Contents of Accumulator A	
		8	1	Stack Pointer – 5	0	Contents of Accumulator B	
		9	1	Stack Pointer – 6	0	Contents of Cond. Code Register	
		10	0	Stack Pointer – 7	1	Irrelevant Data (Note 1)	
		11	1	Vector Address FFFA (Hex)	1	Address of Subroutine (High Order Byte)	
			12	1	Vector Address FFFB (Hex)	1	Address of Subroutine (Low Order Byte)
		RELATIVE					
BCC BHI BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMI BVS	4	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Branch Offset	
		3	0	Op Code Address + 2	1	Irrelevant Data (Note 1)	
		4	0	Branch Address	1	Irrelevant Data (Note 1)	
BSR	8	1	1	Op Code Address	1	Op Code	
		2	1	Op Code Address + 1	1	Branch Offset	
		3	0	Return Address of Main Program	1	Irrelevant Data (Note 1)	
		4	1	Stack Pointer	0	Return Address (Low Order Byte)	
		5	1	Stack Pointer – 1	0	Return Address (High Order Byte)	
		6	0	Stack Pointer – 2	1	Irrelevant Data (Note 1)	
		7	0	Return Address of Main Program	1	Irrelevant Data (Note 1)	
		8	0	Subroutine Address	1	Irrelevant Data (Note 1)	

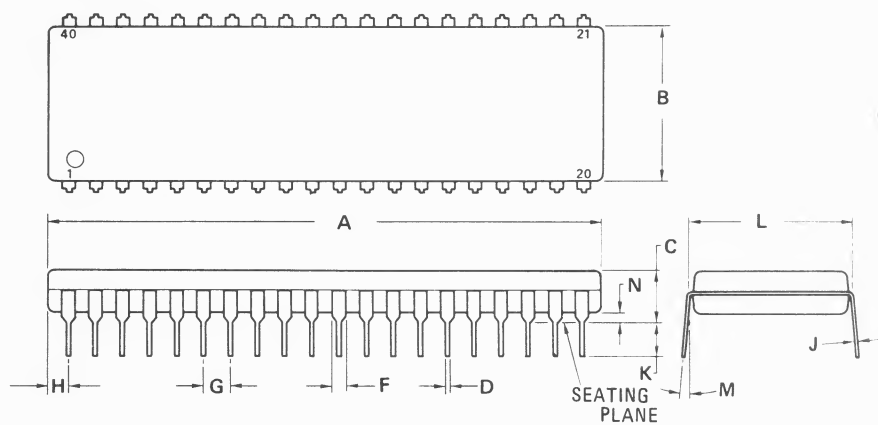
Note 1. If device which is addressed during this cycle uses VMA, then the Data Bus will go to the high impedance three-state condition. Depending on bus capacitance, data from the previous cycle may be retained on the Data Bus.

Note 2. Data is ignored by the MPU.

Note 3. For TST, VMA = 0 and Operand data does not change.

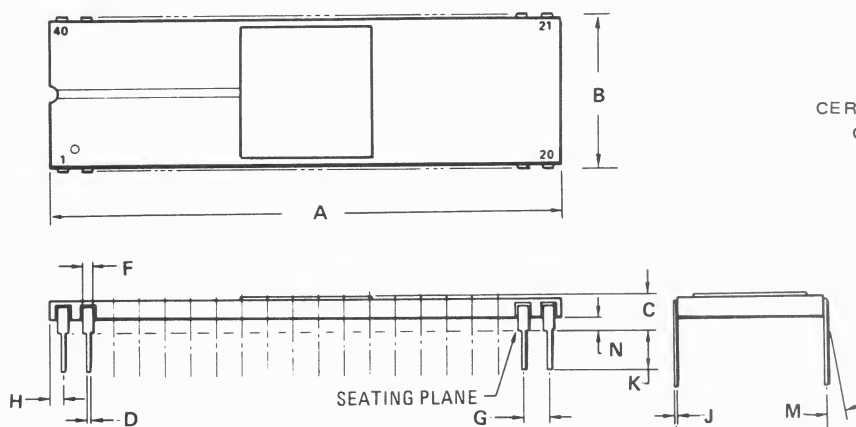
Note 4. While the MPU is waiting for the interrupt, Bus Available will go high indicating the following states of the control lines: VMA is low; Address Bus, R/W, and Data Bus are all in the high impedance state.





P SUFFIX
PLASTIC PACKAGE
CASE 711-01

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.82	52.32	2.040	2.060
B	13.72	14.22	0.540	0.560
C	4.57	5.08	0.180	0.200
D	0.36	0.51	0.014	0.020
F	1.02	1.52	0.040	0.060
G	2.41	2.67	0.095	0.105
H	1.65	2.16	0.065	0.085
J	0.20	0.30	0.008	0.012
K	3.68	4.19	0.145	0.165
L	14.99	15.49	0.590	0.610
M	0°	10°	0°	10°
N	0.51	1.02	0.020	0.040



L SUFFIX
CERAMIC PACKAGE
CASE 715-02

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
M	0°	10°	0°	10°
N	0.51	1.52	0.020	0.060

NOTE:
1. LEADS, TRUE POSITIONED WITHIN
0.25 mm (0.010) DIA (AT SEATING
PLANE), AT MAX. MAT'L
CONDITION.





**MOTOROLA**

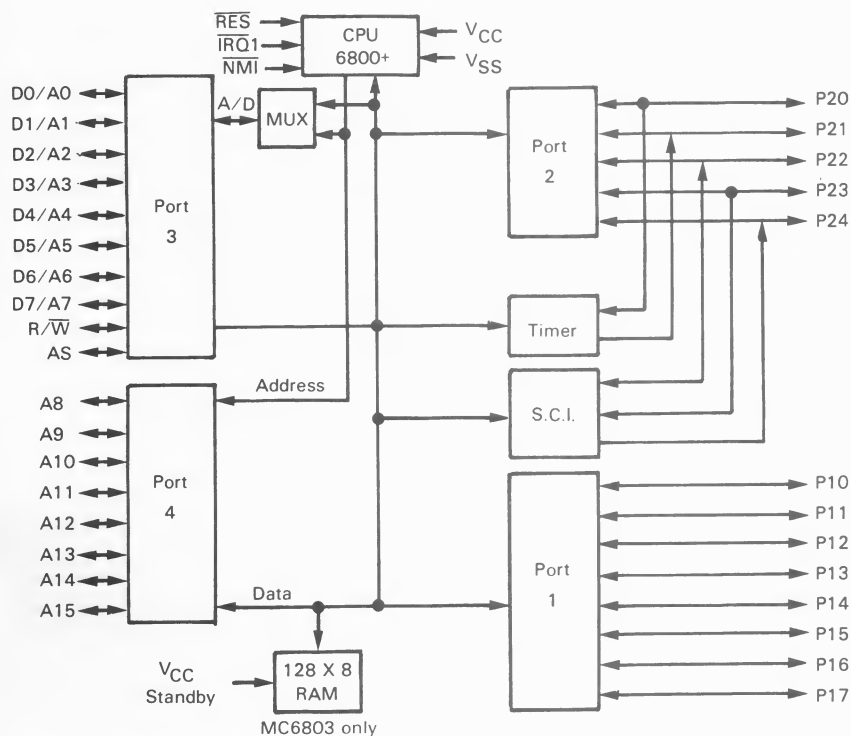
SEMICONDUCTORS

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

Advance Information

The MC6803 is an 8-bit microcomputer which employs a multiplexed address and data system, allowing expandability to 64K words. The MC6803 is object code compatible with the M6800 instruction set and includes improved execution times of key instructions. There are several new 16-bit and 8-bit instructions including an 8 by 8 multiply with 16-bit result. The MC6803 has 128 bytes of RAM, internal clock, SCI, parallel I/O, and three function 16-bit timer all on-board. The MC6803 requires only the addition of a ROM and an external crystal for MCU operation. The MC6803 internal clock's divide by four circuitry allows for use of the inexpensive 3.58 MHz color-burst crystal. The MC6803 MCU is fully TTL compatible and requires only one +5.0 volt power supply. An external RAM is needed with the MC6803 NR.

- Expanded M6800 Instruction Set
- Full Object Code Compatibility With M6800 MPU's
- Multiplexed Address and Data
- Compatible With Existing M6800 Peripherals
- 8 X 8 Multiply With 16-bit Result
- Up to 13 Parallel I/O Lines
- 128 Bytes On-Board RAM on MC6803
- On-Board RAM Retainable With V_{CC} Standby
- Serial Communications Interface On-Board
- 16-bit Timer On-Board
- Internal Clock/Divide by Four Circuitry
- Full TTL Compatibility
- Full Interrupt Capability

FIGURE 1 — BLOCK DIAGRAM

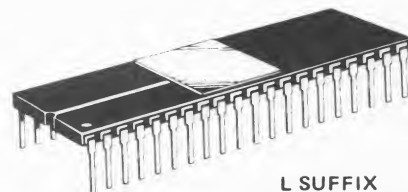
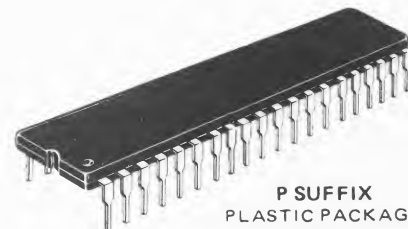
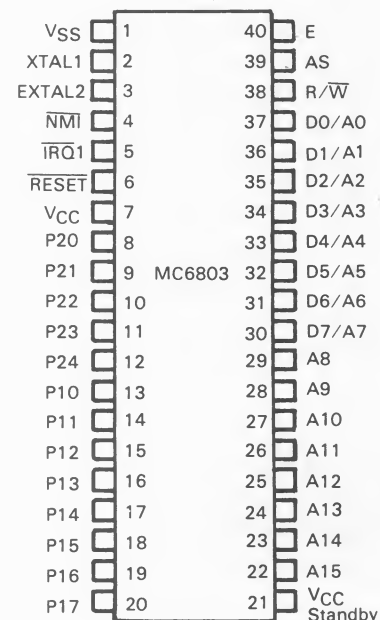
MC6803 MC6803NR

(NO RAM)

MOS

(N-CHANNEL, SILICON-GATE,
DEPLETION LOAD)

MICROPROCESSOR WITH I/O FEATURES

**L SUFFIX**
CERAMIC PACKAGE
CASE 715**P SUFFIX**
PLASTIC PACKAGE
CASE 711**FIGURE 2 — PIN ASSIGNMENT**

ELECTRICAL CHARACTERISTICS ($V_{CC} + 5.0V \pm 5\%$, $V_{SS} = 0$, $T_A = T_L$ to T_H unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Reset	V_{IH}	$V_{SS} + 2.0$ $V_{SS} + 4.0$	—	V_{CC} V_{CC}	Vdc
Input Low Voltage	V_{IL}	$V_{SS} - 0.3$	—	$V_{SS} + 0.8$	Vdc
Three-State (Off State) Input Current P10-P17 ($V_{in} = 0.4$ to 2.4 Vdc) P20-P24, P30-P37	I_{TSI} I_{TSI}	— —	2.0 2.0	10 10	μ Adc μ Adc
Output High Voltage All Outputs ($I_{Load} = -200 \mu$ Adc)	V_{OH}	$V_{SS} + 2.4$	—	—	Vdc
Output Low Voltage All Outputs ($I_{Load} = 1.6$ mAdc)	V_{OL}	—	—	$V_{SS} + 0.4$	Vdc
Power Dissipation	P_D	—	—	1200	mW
Capacitance ($V_{in} = 0$, $T_A = 25^\circ C$, $f = 1.0$ MHz) A0/D0-A7/D7, A8-A15, AS P10-P17, P20-P24 Reset, \overline{IRQ} , R/ \overline{W}	C_{in}	— — —	— — —	12.5 10 7.5	pF
Standby Voltage (Not Operating) (Operating)	V_{SBB} V_{SB}	4.00 4.75	— —	5.25 5.25	Vdc
Frequency of Operation (Input Clock $\div 4$) Crystal	f_{xtal}	0.1 3.5	—	1.0 4.0	MHz MHz

PERIPHERAL TIMING (Figures 3 and 4)

Peripheral Data Setup Time	t_{PDSU}	200	—	—	ns
Peripheral Data Hold Time	t_{PDH}	0	—	—	ns
Delay Time, Enable Positive Transition to OS3 Negative Transition	t_{OSD1}	—	—	350	ns
Delay Time, Enable Positive Transition To OS3 Positive Transition	t_{OSD2}	—	—	350	ns
Delay Time, Enable Negative Transition to Peripheral Data Valid	t_{PWD}	—	—	350	ns
Delay Time, Enable Negative Transition to Peripheral CMOS Data Valid $V_{CC} - 30\% V_{CC}$, P20-P24 (User needs 10k Ω pullup resistor)	t_{CMOS}	—	—	2.0	μ s
Darlington Drive Current $V_O = 1.5$ Vdc, P10–P17	I_{OH}	-1.0	-2.5		mAdc

BUS TIMING (Figure 5)

Characteristic	Symbol	Min	Typ	Max	Unit
Cycle Time	t_{cyc}	1	—	10	μ s
Address Strobe Pulse Width High	PWASH	220	—	—	ns
Address Strobe Rise Time	t_{ASR}	—	—	50	ns
Address Strobe Fall Time	t_{ASF}	—	—	50	ns
Address Strobe Delay Time	t_{ASD}	60	—	—	ns
Enable Rise Time	t_{Er}	—	—	50	ns
Enable Fall Time	t_{Ef}	—	—	50	ns
Enable Pulse Width High Time	PWEH	450	—	—	ns
Enable Pulse Width Low Time	PWEL	450	—	—	ns
Address Strobe to Enable Delay Time	t_{ASED}	60	—	—	ns
Address Delay Time	t_{AD}	—	—	270	ns
Data Delay Time	t_{DDW}	—	—	225	ns
Data Set-Up Time	t_{DSR}	100	—	—	ns
Hold Time — Read	t_{HR}	20	—	100	ns
— Write	t_{HW}	20	—	—	ns
Address Delay Time for Latch	t_{ADL}	—	—	200	ns
Address Hold Time for Latch	t_{AHL}	20	—	—	ns
Address Hold Time	t_{AH}	20	—	—	ns
Total Up Time	t_{ut}	750	—	—	ns



MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	Vdc
Input Voltage	V_{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to 70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C
Thermal Resistance Plastic Package	θ_{JA}	100	°C/W
Ceramic Package		50	

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit. For proper operation it is recommended that V_{in} and V_{out} be constrained to the range $V_{SS} \leq (V_{in} \text{ or } V_{out}) \leq V_{DD}$.

MODE SELECTION TIMING (Figure 6)

Characteristic	Symbol	Min	Typ	Max	Unit
Mode Programming Input Voltage Low	$V_{IL}(MP)$	—	—	2.5	Vdc
Mode Programming Input Voltage High	$V_{IH}(MP)$	4.0	—	—	Vdc
RESET Low Pulse Width	PW_{RSTL}	3.0	—	—	Cycles
Mode Programming Set-Up Time	t_{MPS}	2.0	—	—	Cycles
Mode Programming Hold Time	t_{HMP}	150	—	—	ns

IMPORTANT NOTICE

The MC6803/MC6803NR is one of the finest microprocessors available, and Motorola expects to improve it even further. Only I/C's with date codes 8DXXXX, 1WXXXX and 5GXXXX need the following extra considerations:

1. Be sure software initializes (clears) RDRF, bit 7 in the Transmit/Receive Control and Status register, Address \$0011, before data is input to the Serial Communication Interface (SCI).
2. SCI software should discard a framing error which may occur following a line-break NRZ receive mode only.
3. If an external 8X baud-rate clock is used for the SCI, the external clock should be synchronized before input to port 2, bit 2. A simple D-type flip-flop clocked by the rising edge of E is suitable as is E itself. Avoid aberrations in the duty cycle or frequency of the external clock by limiting the frequency input to the D-flip-flop to $\leq E/2$ Hz.



PORTS 1 & 2

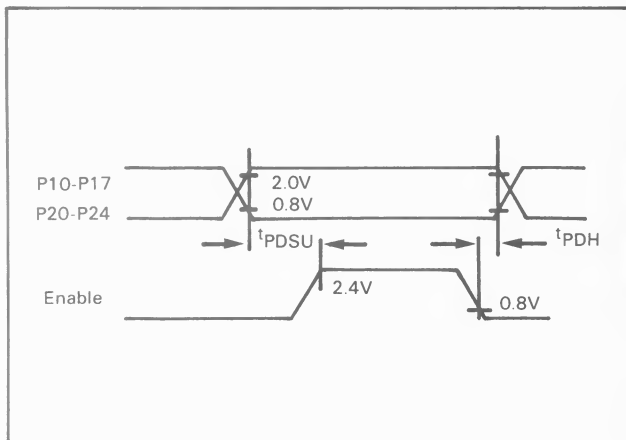
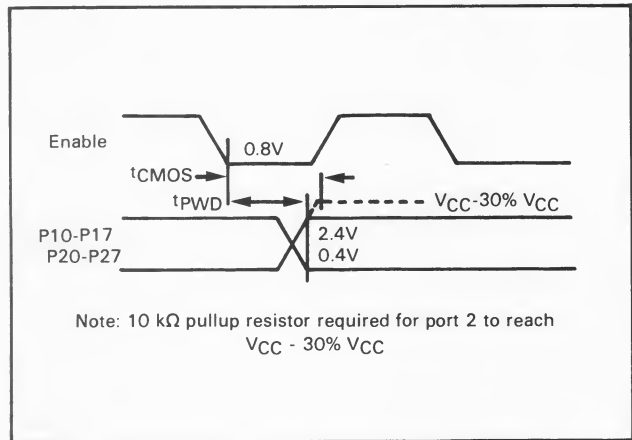
FIGURE 3 — PERIPHERAL DATA SETUP AND HOLD TIMES
(Read Mode)FIGURE 4 — PERIPHERAL DATA DELAY TIMES
(Write Mode)

FIGURE 5 — MULTIPLEXED BUS TIMING

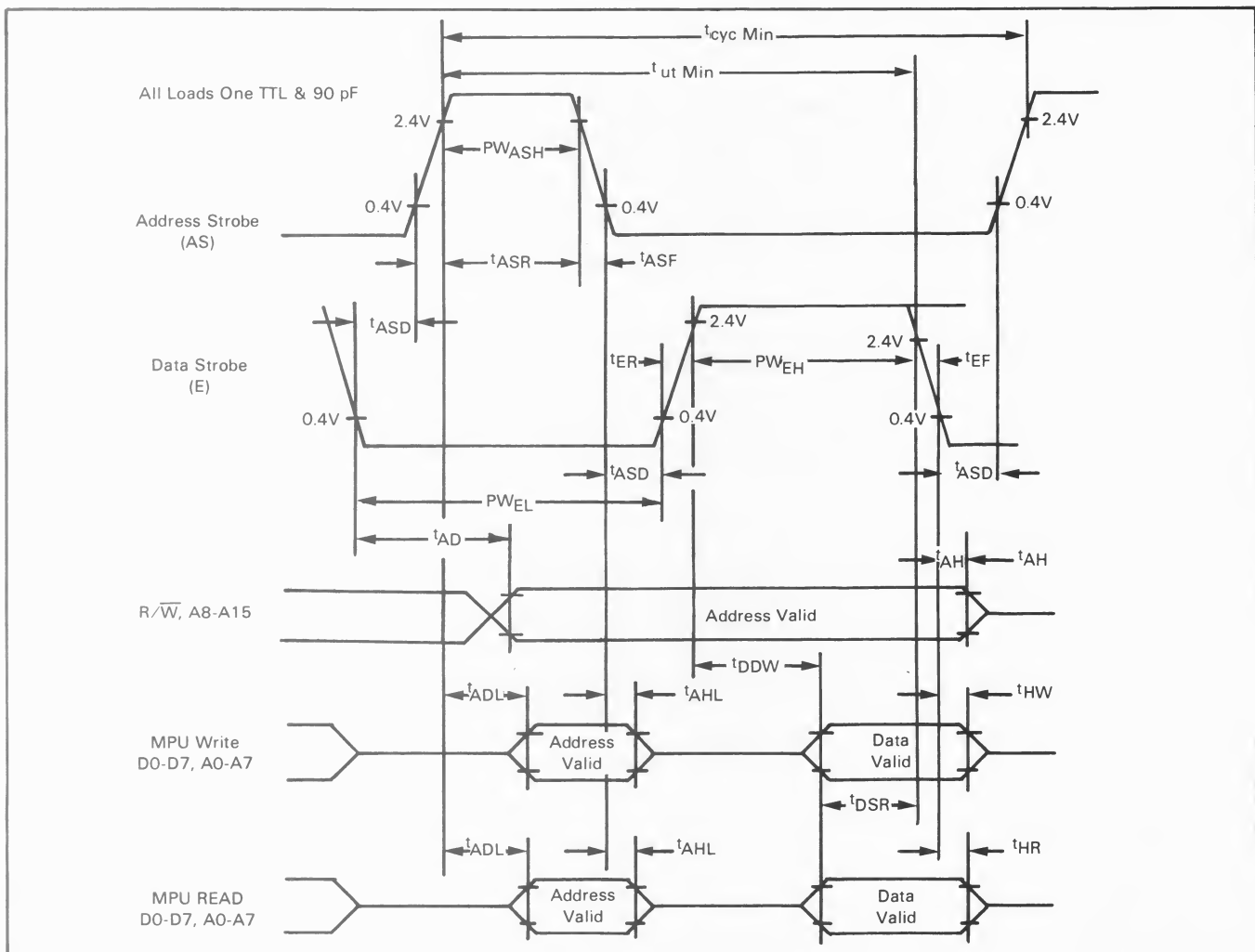


FIGURE 6 — TIMING FOR MODE SELECTION

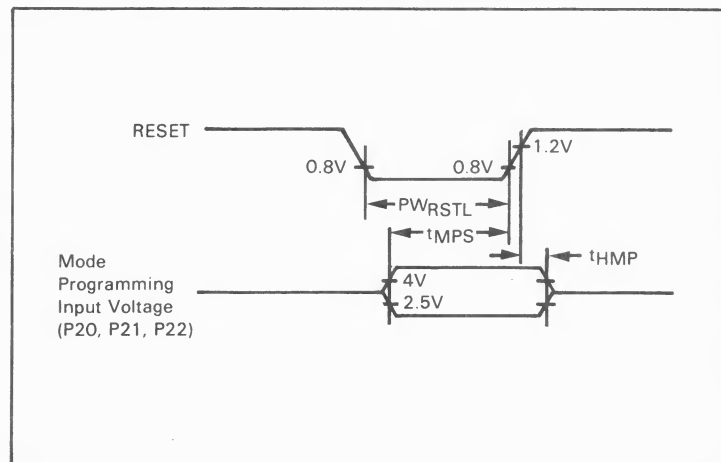


FIGURE 7 — BUS TIMING TEST LOAD FOR A0/D0-A7/D7, AS, A8-A15

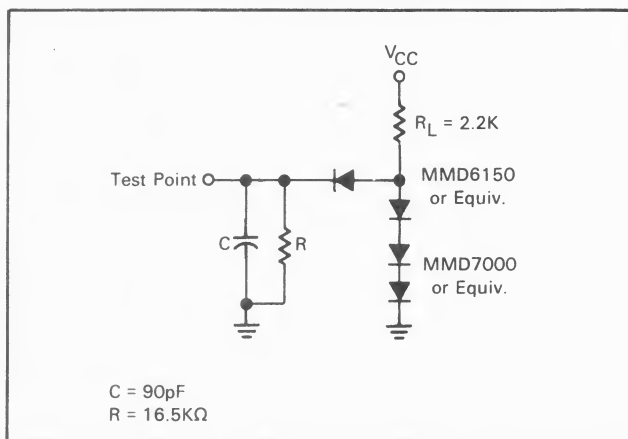


FIGURE 8 — TEST LOAD FOR PORTS 1 AND 2

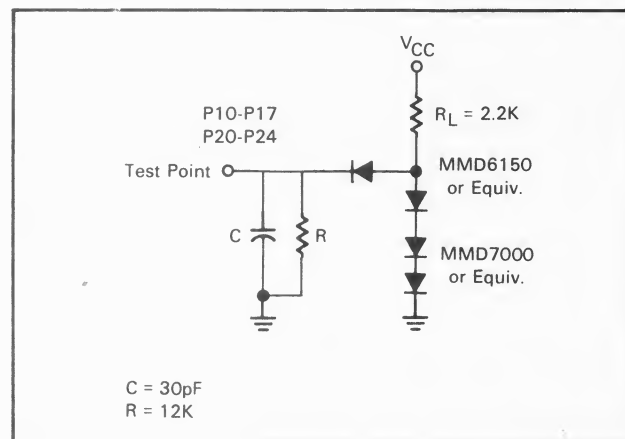


FIGURE 9 — TYPICAL DATA BUS OUTPUT DELAY versus CAPACITIVE LOADING

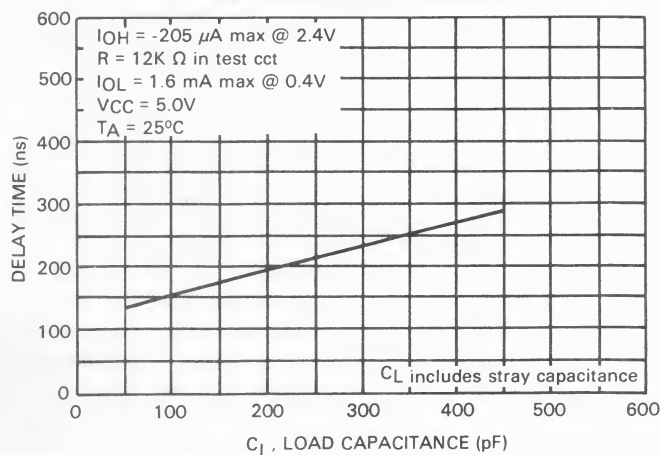
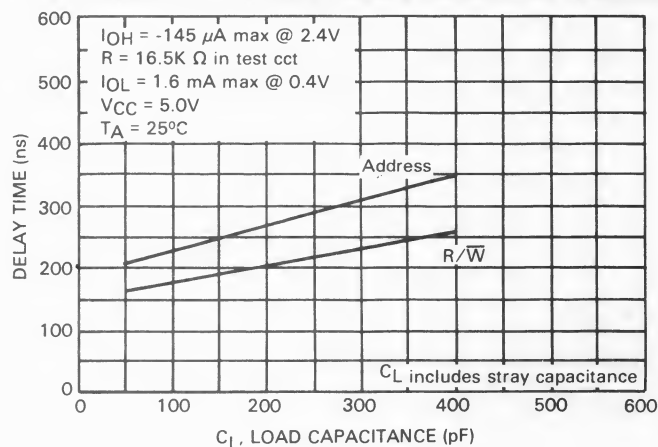


FIGURE 10 — TYPICAL READ/WRITE AND ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING



SIGNAL DESCRIPTIONS

V_{CC} and V_{SS}

These two pins are used to input power and ground to the chip. The voltage supplied should be +5 volts $\pm 5\%$.

XTAL1 and EXTAL2

These connections are for a parallel resonant fundamental crystal, AT-cut. Divide-by-4 circuitry is included with the internal oscillator, so a 4 MHz crystal may be used to run the system at 1 MHz. The divide by 4 circuitry allows for use of the inexpensive 3.58 MHz Color TV crystal. Two 39 pF capacitors are needed from the two crystal pins to ground to insure reliable operation. EXTAL2 may be driven by an external clock source with a 40/60% duty cycle. XTAL1 must be grounded if an external clock is used.

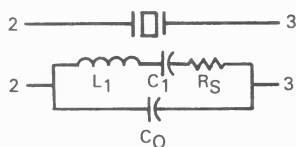
XTAL

The 6801 oscillator is designed to accept parallel resonance crystals, typical AT-cut, in the frequency range of 3.5 to 5.0 MHz. Typical crystal impedance parameters are shown in the following table as referenced to the crystal equivalent circuit in the associated figure.

6801 Crystal Parameters

	3.5 MHz	4.10 MHz	5.0 MHz
RS	60 Ω	50 Ω	30 Ω
C ₀	3.5 pF	6.5 pF	6.5 pF
C ₁	.015 pF	.025 pF	.027 pF
C _L	25 pF	25 pF	25 pF
Q	40K	30K	40K

All Parameters Are $\pm 10\%$

**V_{CC} Standby (MC6803)**

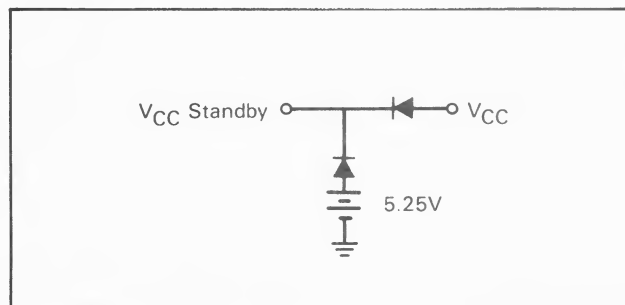
This input pin should supply +5 volts $\pm 5\%$ to the standby RAM on the chip. The first 64 bytes of RAM will be maintained in the power down mode with 8 mA current max. The circuit of Figure 11 can be utilized to assure that V_{CC} Standby does not go below V_{SBB} during power down.

To retain information in the RAM during power down, the following procedure is necessary:

1) Write "0" into the RAM enable bit, RAM E. RAM E is bit 6 of the RAM Control Register at location \$0014. This disables the entire RAM, thereby protecting the standby portion at power down.

2) Keep V_{CC} Standby greater than V_{SBB}.

V_{CC} Standby (MC6803NR) Connect to V_{CC}.

FIGURE 11 — BATTERY BACKUP FOR V_{CC} STANDBY**Reset**

This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. On power up, the reset must be held low for at least 20 ms. During operation, Reset, when brought low, must be held low at least 3 clock cycles.

When a high level is detected, the MPU does the following:

- All the higher order address lines will be forced high.
- I/O Port 2 bits, 2, 1, and 0 are latched into programmed control bits PC2, PC1 and PC0.
- The last two locations in memory (\$FFFE, \$FFFF) will be used for the restart vector.
- The interrupt mask bit is set; must be cleared before the MPU can recognize maskable interrupts.

Enable (E)

This output supplies the external clock for the rest of the system when the internal oscillator is used. It is a single phase, TTL compatible clock, and will be the divide-by-4 result of the crystal frequency. It will drive one TTL load and 90 pF.

Non-Maskable Interrupt (NMI)

A low-going edge on this input requests that a non-maskable-interrupt sequence be executed by the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the $\overline{\text{NMI}}$ signal. The interrupt mask bit in the Condition Code Register has no effect on $\overline{\text{NMI}}$.

In response to an $\overline{\text{NMI}}$ interrupt, the Index Register, Program Counter, Accumulators, and Condition Code Register are stored on the stack. At the end of the sequence, a 16-bit vector address located in memory locations \$FFFC and \$FFFD is used to cause the MPU to branch to an $\overline{\text{NMI}}$ interrupt service routine.

A 3.3k Ω external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

Inputs $\overline{\text{IRQ1}}$ and $\overline{\text{NMI}}$ are hardware interrupt lines that are sampled during E, and will start the interrupt routine on the $\overline{\text{E}}$ following the completion of an instruction.



Interrupt Request ($\overline{\text{IRQ1}}$)

This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further maskable interrupts may occur. At the end of the cycle, a 16-bit vectoring address which is located in memory locations \$FFF8 and \$FFF9 is used to cause the MPU to branch to an $\overline{\text{IRQ}}$ interrupt routine in memory.

The $\overline{\text{IRQ1}}$ requires a 3.3k Ω external resistor to V_{CC} which should be used for wire-OR and optimum control of interrupts. Internal Interrupts will use an internal interrupt line ($\overline{\text{IRQ2}}$). This interrupt will operate the same as $\overline{\text{IRQ1}}$ except that it will use the vector address of \$FFF0 thru \$FFF7. $\overline{\text{IRQ1}}$ will have priority over $\overline{\text{IRQ2}}$ if both occur at the same time. The Interrupt Mask Bit in the condition mode register masks both interrupts. Refer to Figure 12 for Interrupt Vector Map.

FIGURE 12 — MEMORY MAP FOR INTERRUPT VECTORS

	Vector		Description
	MS	LS	
Highest Priority	FFFE,	FFFF	Restart
	FFFC,	FFFD	Non-Maskable Interrupt
	FFFA,	FFFB	Software Interrupt
	FFF8,	FFF9	$\overline{\text{IRQ1}}$
	FFF6,	FFF7	$\overline{\text{IRQ2}}$ /Timer Input Capture
	FFF4,	FFF5	$\overline{\text{IRQ2}}$ /Timer Output Compare
	FFF2,	FFF3	$\overline{\text{IRQ2}}$ /Timer Overflow
Lowest Priority	FFF0,	FFF1	$\overline{\text{IRQ2}}$ /Serial I/O Interrupt

Address Strobe (AS)

This output is used to latch the 8 LSB's of address which are multiplexed with data. An 8-bit latch is utilized in conjunction with Address Strobe, as described in Figure 13. Address Strobe signals the time to latch the address so the lines can output data during the E pulse. The timing for this signal is shown in the MC6803 Bus Timing Figure 5. This signal is also used to disable the address from the multiplexed bus allowing a deselect time T_{ASD} before the data is enabled to the bus.

Read/Write ($\overline{\text{R/W}}$)

This TTL compatible output indicates to the peripherals and memory devices that the MPU is in a Read (high) or a Write (low) state. The normal state of this signal is Read (high). This output is capable of driving one TTL load and 90 pF.

Address/Data Bus (A0/D0-A7/D7)

Eight pins are used for the multiplexed address and data bus. These pins provide the lower order address lines plus the 8-bit bidirectional data bus. The outputs are TTL compatible and can drive 1 TTL load and 90 pF.

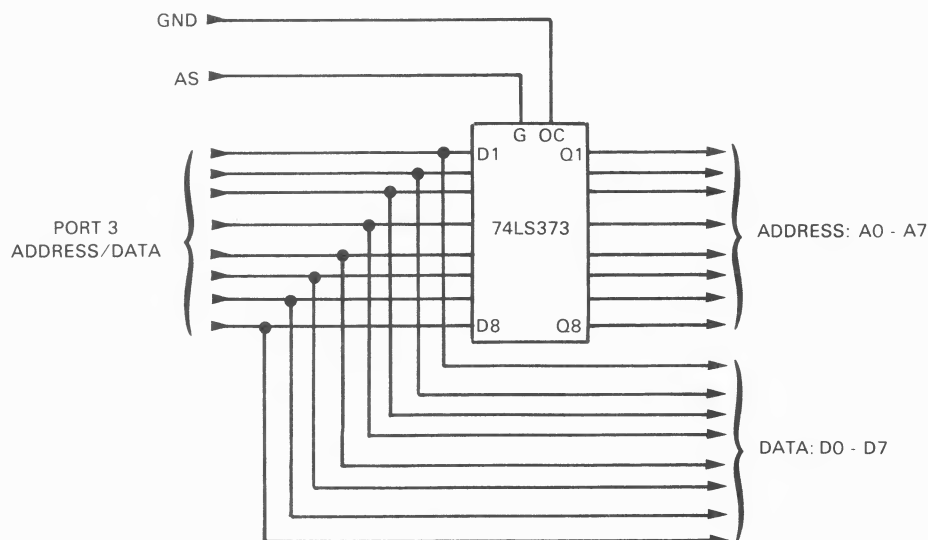
An external latch may be supplied by the user in order to supply the full 16-bit non-multiplexed address lines. The latch is controlled by the address strobe.

The SN74LS373 Transparent octal D-type latch can be used with MC6803/MC6803NR to latch the least significant address byte. Figure 13 shows how to connect the latch to MC6803.

Address Bus (A8-A15)

These eight lines output the higher order address lines allowing for the full 64K word expandability. Timing is shown in Figure 5. The address bus outputs are TTL compatible and can drive 1 TTL load and 90 pF.

FIGURE 13 — LATCH CONNECTION



FUNCTION TABLE

OUTPUT CONTROL	ENABLE G	D	OUTPUT Q
L	H	H	H
L	H	L	L
L	L	X	Q_0
H	X	X	Z



PERIPHERAL INTERFACE LINES

The MC6803/MC6803NR provides an 8-bit port and a 5-bit port for interfacing to peripheral devices. They are bidirectional in that each bit can be programmed as either an input or an output by writing to the associated bit in the port's Data Direction Register. A "1" in the corresponding Data Direction Register bit will cause that I/O line to be an output. A "0" in the corresponding Data Direction Register bit will cause the I/O line to be an input. There is one exception in Port 2. Bit "1" of Port 2 can be either an input line or the timer output, which precludes its use as an output for any other purpose. The two ports and their associated Data Direction Registers are addressed as follows:

Ports	Port Address	Data Direction Register Address
I/O Port 1	\$0002	\$0000
I/O Port 2	\$0003	\$0001

I/O Port 1 (P10-P17)

This is an 8-bit port whose individual bits may be defined as inputs or outputs by its data direction register. The 8 input buffers have 3-state capability allowing them to enter a high impedance state when the peripheral data lines are used as inputs. In order to be read properly, the voltage on the input lines must be greater than 2.0V for a

logic 1 and less than .8V for a logic 0. As outputs, these lines are TTL compatible and may also be used as a source of up to 1mA at 1.15V to directly drive a Darlington base. After Reset, the I/O lines are configured as inputs.

I/O Port 2 (P20-P24)

This port has five lines whose individual bits may be defined as inputs or outputs by its associated data direction register. Bit "1" can be selected as an input or timer output. The 5 input buffers have three-state capability allowing them to enter a high impedance state when used as inputs. In order to be read properly, the voltage on the input lines must be greater than 2.0V for a logic 1 and less than .8V for a logic 0. As outputs, this port has no internal pullup resistors but will drive TTL inputs directly. For driving CMOS inputs, an external pullup resistor (10K) is required. After Reset, the I/O lines are configured as inputs.

Port 2 also provides access to the Serial Communications Interface and the Timer. The Timer has two associated lines: Timer Input (P20) and Timer Output (P21). The Serial Communications Interface has three associated lines: Transmitter (P24), Receiver (P23) and Clock (P22). Both the Timer and SCI have associated control registers which allow for their selection and access on the I/O Port 2 lines.

POWER-UP AND RESET HARDWARE

In order for the chip to operate properly after Reset, three pins on Port 2 (pins 10, 9 and 8) must be held to a certain level during Reset. If the user has an MC6803, the circuit of Figure 14 may be utilized. In any event, pins 10, 9 and 8 must be held low, high, low respectively during Reset.

FIGURE 14 — DIODE CONFIGURATION FOR MC6803

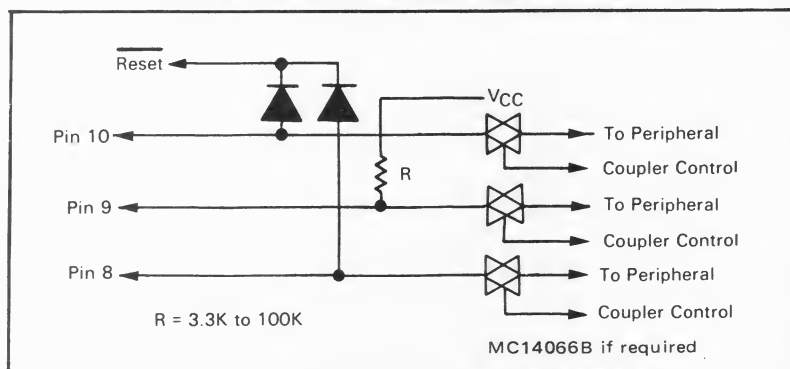
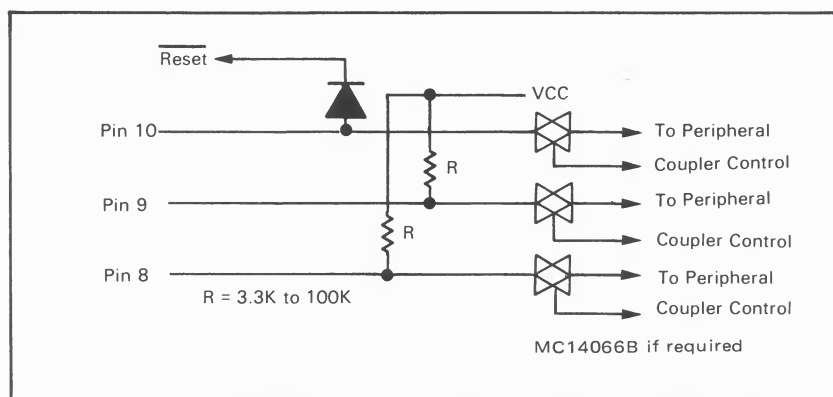


FIGURE 15 — DIODE CONFIGURATION FOR MC6803NR



For the MC6803NR, the configuration of Figure 15 may be used. Pins 10, 9, and 8 must be held low, high, high respectively during Reset.

If pins 10, 9 and 8 go to devices which may input signals

differing from the signals required during reset, couplers or three-state buffers are necessary.

The MC14066B can be used to provide this isolation between the peripheral device and the MCU during reset.

PROGRAMMABLE TIMER

The MC6803/MC6803NR contains an on-chip 16-bit programmable timer which may be used to perform measurements on an input waveform while independently generating an output waveform. Pulse widths for both input and output signals may vary from a few microseconds to many seconds. The timer hardware consists of:

- an 8-bit control and status register,
- a 16-bit free running counter,
- a 16-bit output compare register, and
- a 16-bit input capture register

A block diagram of the timer registers is shown in Figure 16.

Free Running Counter (\$0009:000A)

The key element in the programmable timer is a 16-bit free-running counter which is incremented by the MPU clock. The counter value may be read by the MPU software at any time. The counter is cleared to zero on **RESET** and may be considered a read-only register with one exception. Any MPU write to the counter's address (\$09) will always result in a preset value of \$FFF8 being loaded into the counter regardless of the value involved in the write. This preset feature is intended for testing operation of the part, but may be of value in some applications. However, this will also adversely affect operation of the SCI.

Output Compare Register (\$000B:000C)

The Output Compare Register is a 16-bit read/write register which is used to control an output waveform. The contents of this register are constantly compared with the current value of the free running counter. When a match is found, a flag is set (OCF) in the Timer Control and Status Register (TCSR) and the current value of the Output Level bit (OLVL) in the TCSR is clocked to the output level register. Providing the Data Direction Register for Port 2 Bit 1 contains a "1" (output), the output level register value will appear on the pin for Port 2 Bit 1. The values in

the Output Compare Register and Output level bit may then be changed to control the output level on the next compare value. The Output Compare Register is set to \$FFFF during **RESET**. The Compare function is inhibited for one cycle following a write to the high byte of the Output Compare Register to insure a valid 16-bit value is in the register before a compare is made.

Input Capture Register (\$000D:000E)

The Input Capture Register is a 16-bit read-only register used to store the current value of the free running counter when the proper transition of an external input signal occurs. The input transition change required to trigger the counter transfer is controlled by the Input Edge bit (IEDG) in the TCSR. The Data Direction Register bit for Port 2 Bit 0, should* be cleared (zero) in order to gate in the external input signal to the edge detect unit in the timer.

*With Port 2 Bit 0 configured as an output and set to "1", the external input will still be seen by the edge detect unit.

Timer Control and Status Register (TCSR) (\$0008)

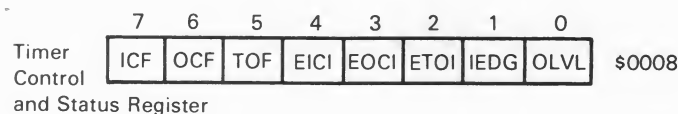
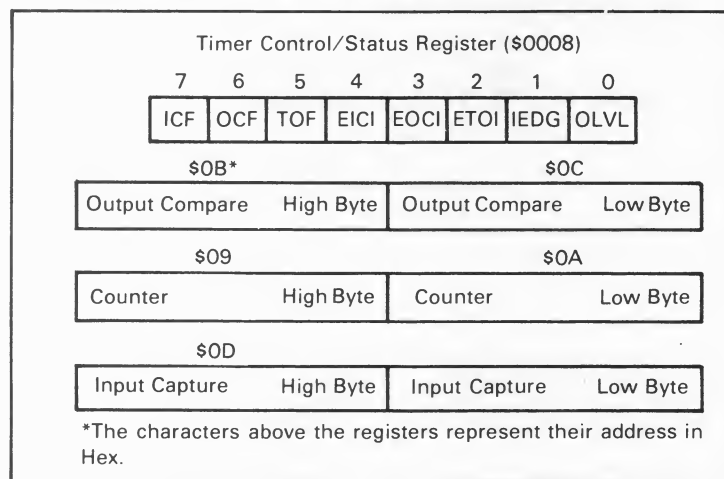
The Timer Control and Status Register consists of an 8-bit register of which all 8 bits are readable but only the low order 5 bits may be written. The upper three bits contain read-only timer status information and indicate that:

- a proper transition has taken place of the input pin with a subsequent transfer of the current counter value to the input capture register, (bit 7).
- a match has been found between the value in the free running counter and the output compare register, (bit 6).
- when the free running counter has overflowed from \$FFFF to \$0000 (bit 5).

Each of the flags may be output to the MC6803 internal Interrupt Request (IRQ2) with an individual Enable bit in the TCSR. If the I-bit in the MC6803/MC6803NR Condition Code register is clear, a priority vectored interrupt will occur corresponding to the flag bit(s) set. A description for each bit follows:



FIGURE 16 — BLOCK DIAGRAM OF TIMER REGISTERS



Bit 0	OLVL	Output Level — This value is clocked to the output level register on an output compare. If the DDR for Port 2 bit 1 is set, the value will appear on the output pin.	Bit 4	EICI	Enable Input Capture Interrupt — When set , this bit enables IRQ2 to occur on the internal bus for an input capture interrupt; when clear the interrupt is inhibited.
Bit 1	IEDG	Input Edge — This bit controls which transition of an input will trigger a transfer of the counter to the input capture register. The DDR for Port 2 bit 0 must be clear for this function to operate. IEDG = 0 Transfer takes place on a negative (high-to-low transition). IEDG = 1 Transfer takes place on a positive edge (low-to-high transition).	Bit 5	TOF	Timer Overflow Flag — This read-only bit is set when the counter contains \$0000. It is cleared by a read of the TCSR (with TOF set) followed by an MPU read of the Counter (\$09).
Bit 2	ETOI	Enable Timer Overflow Interrupt — When set , this bit enables IRQ2 to occur on the internal bus for a TOF interrupt; when clear the interrupt is inhibited.	Bit 6	OCF	Output Compare Flag — This read-only bit is set when a match is found between the output compare register and the free running counter. It is cleared by a read of the TCSR (with OCF set) followed by an MPU write to the output compare register (\$0B or \$0C).
Bit 3	EOCI	Enable Output Compare Interrupt — When set , this bit enables IRQ2 to appear on the internal bus for an input capture; when clear the interrupt is inhibited.	Bit 7	ICF	Input Capture Flat — This read-only status bit is set by a proper transition on the input to the edge detect unit; it is cleared by a read of the TCSR (with ICF set) followed by an MPU read of the Input Capture Register (\$0D).

SERIAL COMMUNICATIONS INTERFACE

The MC6803 contains a full-duplex asynchronous serial communications interface (SCI). Two serial data formats (standard mark/space (NRZ) or Bi-phase) are provided at several different data rates. The controller comprises a transmitter and a receiver which operate independently of each other but in the same data format and at the same data rate. Both transmitter and receiver communicate with the MPU via the data bus, and with the outside world via bits 2, 3, and 4 of Port 2.

Wake-Up Feature

In a typical multi-processor application, the software protocol will usually contain a destination address in the initial byte(s) of the message. In order to permit non-selected MPU's to ignore the remainder of the message, a wake-up feature is included whereby all further SCI Receive interrupts may be optionally inhibited until the data line goes idle. The "wake-up" bit is automatically reset by a string of ten consecutive 1's which indicates an idle data line. The software protocol must provide for the short idle period between consecutive messages and no idle period within messages.

Programmable Options

The following features of the MC6803 serial I/O section are programmable:

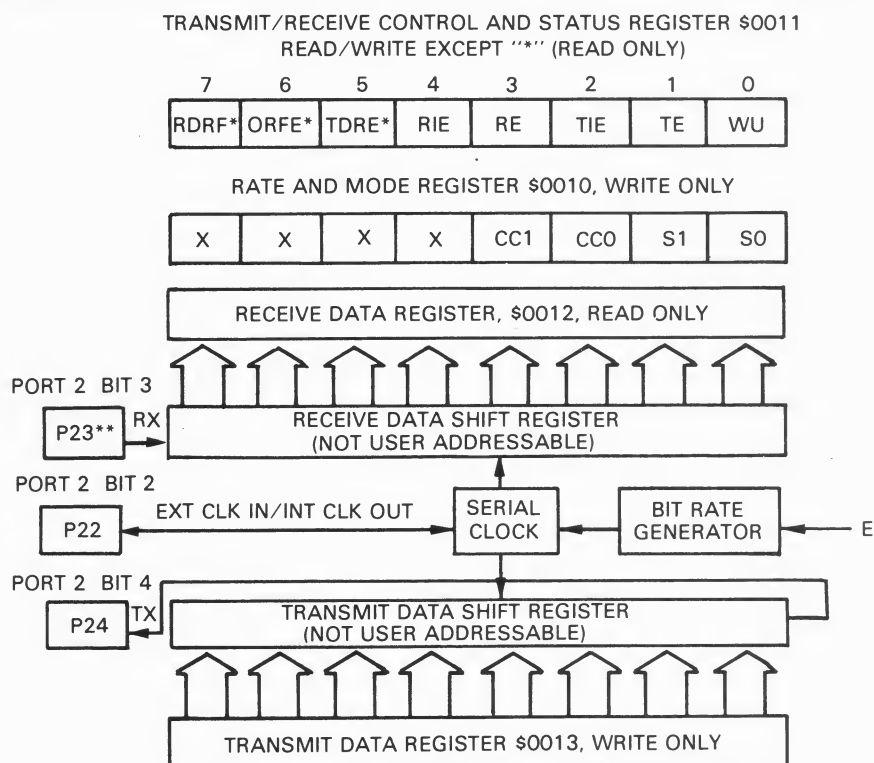
- format — standard mark/space (NRZ) or Bi-phase
- clock — external or internal
- baud rate — one of 4 per given MPU $\phi 2$ clock frequency, or external clock X8 input
- wake-up feature — enabled or disabled
- interrupt requests — enabled individually for transmitter and receiver data registers.
- clock output — internal clock enabled or disabled to Port 2 (bit 2)
- Port 2 (bits 3 and 4) — dedicated or not dedicated to serial I/O individually for transmitter and receiver.

Serial Communications Hardware

The serial communications hardware is controlled by 4 registers as shown in Figure 17. The registers include:

- an 8-bit control and status register
 - a 4-bit write only rate and mode control register
 - an 8-bit read-only receive data register and
 - an 8-bit write-only transmit data register.
- In addition to the four registers, the serial I/O section utilizes bit 3 (serial input) and bit 4 (serial output) of Port 2. Bit 2 of Port 2 is utilized if either the internal-clock-out or external-clock-in options are selected.

FIGURE 17 — SERIAL I/O REGISTERS



**P23 refers to Port 2, bit 3.

NOTE: (LSB First)



Transmit/Receive Control and Status (TRCS) Register

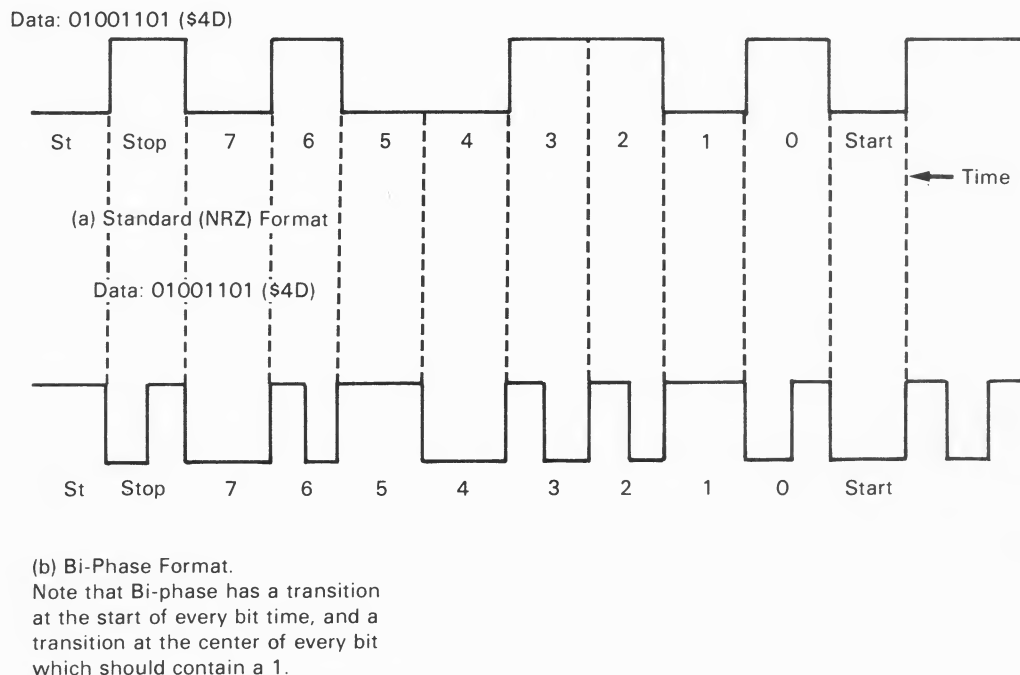
The TRCS register consists of an 8-bit register of which all 8 bits may be read while only bits 0-4 may be written. The register is initialized to \$20 on **RESET**. The bits in the TRCS registers are defined as follows:

7	6	5	4	3	2	1	0	
RDRF	ORFE	TDRE	RIE	RE	TIE	TE	WU	ADDR: \$0011

- | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Bit 0 WU "Wake-up" on Idle Line — when set, enables wake-up function; cleared by hardware on receipt of ten consecutive 1's. WU will not set if the line is idle.</p> <p>Bit 1 TE Transmit Enable — when set, changes DDR value for Port 2 bit 4 to a "1"; the DDR value cannot be cleared while TE is set and will be left at "1" when TE is subsequently cleared. A preamble of nine consecutive 1's is produced when TE is changed from clear to set. While TE is set, the transmitter output is gated to Port 2 bit 4.</p> <p>Bit 2 TIE Transmit Interrupt Enable — when set, will permit an IRQ2 interrupt to occur if TDRE is set; when clear, the interrupt is inhibited.</p> <p>Bit 3 RE Receiver Enable — when set, changes the DDR value for Port 2 bit 3 to a "0"; DDR value cannot be set while RE is set and will be left at "0" if RE is subsequently cleared. While RE is set, Port 2 bit 3 is gated to the receiver.</p> <p>Bit 4 RIE Receiver Interrupt Enable — when set, will permit an IRQ2 interrupt to occur when either RDRF or ORFE is set; when clear, the interrupt is inhibited.</p> <p>Bit 5 TDRE Transmit Data Register Empty — set by hardware when a transfer is made from the transmit data register to the output shift register. This transfer is synchronized with the bit rate clock. The TDRE bit</p> | <p>is cleared by reading the status register, then writing a new byte into the transmit data register. No data will be transmitted if TDRE is not cleared. TDRE is initialized to 1 by RESET.</p> <p>Bit 6 ORFE Over-Run-Framing Error — set by hardware when an overrun or framing error occurs (receiver only). An overrun is defined as a new byte ready for the Receiver Data Register with the RDRF flag set. A framing error has occurred when the byte boundaries in the bit stream are not synchronized to bit counter. An overrun may be distinguished from a framing error by the corresponding value of RDRF. If RDRF = ORFE = 1, then an overrun has occurred. If RDRF = 0 and ORFE = 1, a framing error has been detected. A new byte is not transferred to the Receive Data Register if RDRF is already set. The ORFE bit is cleared by reading the status register, then the Receive Data Register, or by RESET.</p> <p>Bit 7 RDRF Receive Data Register Full — set by hardware when a transfer from the input shift register to the receive data register is made. The RDRF bit is cleared by reading the status register, then reading the Receive Data Register or by RESET.</p> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



FIGURE 18 — SCI DATA FORMATS



Rate and Mode Control Register

The Rate and Mode Control register controls the following serial I/O variables:

- Baud rate
- format
- clocking source, and
- Port 2 bit 2 configuration

The register consists of 4 bits all of which are write-only and cleared on **RESET**. The 4 bits in the register may be considered as a pair of 2-bit fields. The two low order bits control the bit rate for internal clocking and the remaining two bits control the format and clock select logic. The register definition is as follows:

7	6	5	4	3	2	1	0	
X	X	X	X	CC1	CC0	S1	S0	ADDR: \$0010

Bit 0 **S0** Speed Select — These bits select the Baud rate for the internal clock. The four rates which may be selected are a function of the MPU $\phi 2$ clock frequency. Table 1 lists the available Baud rates.

Bit 1 **S1**

Bit 2 **CC0** Clock Control and Format Select — this 2-bit field controls the format and clock select logic. Table 2 defines the bit field. If external clocking is selected (CC1 = CC0 = 1), the speed select bits are ignored.

Bit 3 **CC1**

TABLE 1 — SCI BIT TIMES & BAUD RATES

Rate Control S1, S0	XTAL \rightarrow $\phi 2$	4.0 MHz	4.9152 MHz*	2.4576 MHz
		1.0 μ s/1.0 MHz	814 ns/1.2288 MHz	1.63 μ s/614.4 KHz
0 0	$\div 16$	16 μ s/62,500 Baud	13.0 μ s/76,800 Baud	26 μ s/38,400 Baud
0 1	$\div 128$	128 μ s/7812.5 Baud	104.2 μ s/9,600 Baud	208 μ s/4,800 Baud
1 0	$\div 1024$	1.024ms/976.6 Baud	833.3 μ s/1,200 Baud	1.67ms/600 Baud
1 1	$\div 4096$	4.096ms/244.1 Baud	3.33ms/300 Baud	6.67ms/150 Baud

*the 68A03 should be used if this frequency is desired



TABLE 2 — SCI FORMAT AND CLOCK SOURCE CONTROL

CC1, CC0	Format**	Clock Source	Port 2 Bit 2
00	Bi-Phase	Internal	Not Used
01	NRZ	Internal	Not Used
10	NRZ	Internal	Output*
11	NRZ	External	Input

*Clock output is available regardless of values for bits RE and TE.

**The formats are illustrated in Figure 18.

Internally Generated Clock

If the user desires the SCI to output a clock, the following requirements are applicable:

- the values of RE and TE are immaterial.
- CC1, CC0 must be set to 10.
- the maximum clock rate will be $E \div 16$ ($S1 = S0 = 0$)
- the clock will be at 1X the bit rate and will have a rising edge at mid-bit.

Externally Generated Clock

If the user desires to input an external clock for the SCI, the following requirements are applicable:

- the CC1, CC0, field in the Rate and Mode Control Register must be set to 11.
- the external clock must be 8 times (X8) the desired baud rate and
- the maximum external clock frequency is $\leq E$.

SERIAL OPERATIONS

The serial I/O hardware must be initialized prior to operation. This sequence will normally consist of:

- writing the desired control bits to the Rate and Mode Control Register and
- writing the desired control bits to the Transmit/Receive Control Register.

The Transmitter Enable (TE) and Receiver Enable (RE) bits may be left set for dedicated operations.

Transmit Operations

The transmit operation is enabled by the TE bit in the Transmit/Receive Control and Status Register. This bit, when set, gates the output of the serial transmit shift register to Port 2 Bit 4 and takes unconditional control over the Data Direction Register value for Port 2, Bit 4.

Setting the TE bit initiates the serial output by first transmitting a ten-bit preamble of 1's. Following the preamble, internal synchronization is established and the transmitter section is ready for operation.

At this point one of two situations exist:

- a) if the Transmit Data Register is empty ($TDRE = 1$), a continuous string of ones will be sent indicating an idle line, or
- b) if data has been loaded into the Transmit Data Register ($TDRE = 0$), the word is transferred to the output shift register and transmission of the data word will begin.

During the serial transfer, the 0 start bit is transmitted. Then the 8 data bits (beginning with bit 0) followed by the stop bit, are transmitted. The TDRE flag bit is set by SCI

hardware when the data transfer from the Transmit Data Register to the Transmit Output Shift Registers occurs.

If software fails to respond to the flag within the proper time, (TDRE is still set when the next normal transfer from the parallel data register to the serial output register should occur) then a 1 will be sent (instead of a 0) at "start" bit time, followed by more 1's until more data is written to the data register. No 0's will be sent while TDRE remains a 1.

Receive Operation

The receive operation is enabled by the RE bit which gates in the serial input from Port 2 Bit 3. The receiver section operation is controlled by the contents of the Transmit/Receive Control and Status Register and the Rate and Mode Control Register.

The data bit interval is divided into 8 sub-intervals for internal synchronization. The received bit stream is synchronized by the first 0 (space) encountered following an idle line condition (continuous ones on the data line).

The approximate center of each bit time is strobed during the next 10 bits. If the tenth bit is not a 1 (stop bit) a framing error is assumed, and bit ORFE is set. If the tenth bit is a 1, the data is transferred to the Receiver Data Register, and receiver flag RDRF is set. If RDRF is still set at the next tenth bit time, ORFE will be set, indicating an over-run has occurred. Both RDRF and ORFE are cleared by reading the TRCS register followed by reading the Receiver Data Register.



RAM CONTROL REGISTER (\$0014)

This register, which is addressed at \$0014, gives status information about standby RAM operations. A 0 in the RAM enable bit (RAME) will disable the RAM, thereby protecting it at power down if V_{CC} Standby is held greater than V_{SBBL} volts.

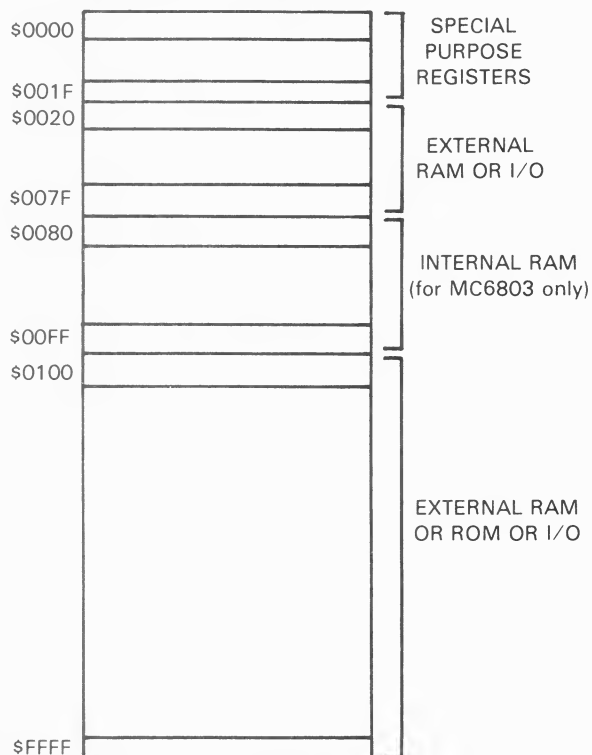
	7	6	5	4	3	2	1	0
\$0014	STANDBY PWR	RAME	X	X	X	X	X	X

Bits 0-5 are not used

- Bit 6 The RAM ENABLE control bit allows the user the ability to disable the RAM. This bit is set to a logic "one" by RESET which enables the RAM and can be written to one or zero under program control. When the RAM is disabled, (logic "zero") data is read from external memory.
- Bit 7 The STANDBY PWR bit is cleared whenever the standby voltage decreases to below V_{SBBL} . This bit is a read/write status flag that may be used to detect whether V_{CC} Standby has decreased to V_{SBBL} . The bit may be set by software, but it is not affected by RESET.

The MC6803 addresses up to 64K bytes of memory for program and/or data storage. The memory map is shown in Figure 19.

FIGURE 19 — MEMORY MAP



MC6803

Locations \$0020 through \$007F access external RAM or I/O. Internal RAM is accessed at \$0080 through \$00FF. However, if the user desires to access external RAM at those locations he may do so by clearing the RAM ENABLE control bit of the RAM Control Register. In this way an extra 128 bytes of external RAM may be used. The first 64 bytes of the 128 bytes of on-chip RAM are provided with a separate power supply. This will maintain the 64 bytes of RAM in the power down mode as explained in the pin description for V_{CC} Standby.

MC6803NR

The first 32 locations are for internal registers as shown in Table 3. The reset, up to 64K, are for external RAM or ROM I/O.

TABLE 3 — SPECIAL REGISTERS

Hex Address	Register
00	Data Direction Port 1
01	Data Direction Port 2
02	I/O Port 1
03	I/O Port 2
08	Timer Control And Status Register
09	Counter High Byte
0A	Counter Low Byte
0B	Output Compare High Byte
0C	Output Compare Low Byte
0D	Input Capture High Byte
0E	Input Capture Low Byte
10	Serial Rate and Mode Register
11	Serial Control and Status Register
12	Serial Receiver Data Register
13	Serial Transmit Data Register
14	RAM Control Register
15-1F	Reserved



GENERAL DESCRIPTION OF INSTRUCTION SET

The MC6803 is upward object code compatible with the MC6800 as it implements the full M6800 instruction set. The execution times of key instructions have been reduced to increase throughput. In addition, new instructions have been added; these include 16-bit operations and a hardware multiply.

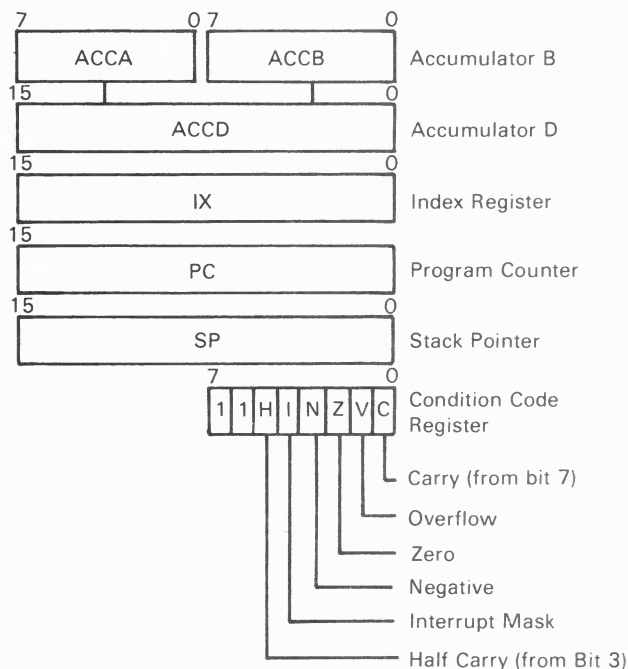
Included in the instruction set section are the following:

- MPU Programming Model (Figure 20)
- Addressing modes
- Accumulator and memory instructions — Table 4
- New Instructions
- Index register and stack manipulations — Table 5
- Jump and branch instructions — Table 6
- Condition code register manipulation instructions — Table 7
- Instruction Execution cycle times — Table 8
- Special operations — Figure 21
- Summary of cycle by cycle operation — Table 9

MPU Programming Model

The programming model for the MC6803/MC6803NR is shown in Figure 20. The double (D) accumulator is physically the same as the A Accumulator concatenated with the B Accumulator so that any operation using accumulator D will destroy information in A and B.

FIGURE 20 — MCU PROGRAMMING MODEL



MPU Addressing Modes

The MC6803 eight-bit microcomputer unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 4 along with the associated instruction execution time that is given in machine cycles. With a XTAL frequency of 4 MHz, these times would be microseconds.

Accumulator (ACCX) Addressing — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

Immediate Addressing — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MCU addresses this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

Direct Addressing — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

Extended Addressing — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight bits of the address of the operand. The third byte of the instruction is used as the lower eight bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

Indexed Addressing — In indexed addressing, the addresses contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

Inherent Addressing — In the inherent addressing mode, the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

Relative Addressing — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.



TABLE 4 — ACCUMULATOR & MEMORY INSTRUCTIONS

ACCUMULATOR AND MEMORY		IMMED.		DIRECT		INDEX		EXTEND		INHERENT		Boolean/ Arithmetic Operation	COND. CODE					
Operations	Mnemonic	OP	~ #	OP	~ #	OP	~ #	OP	~ #	OP	~ #		5	4	3	2	1	0
Add	ADDA	8B	2 2	9B	3 2	AB	4 2	BB	4 3			$A + M \rightarrow A$	↑	●	↑	↑	↑	↑
	ADDB	CB	2 2	DB	3 2	EB	4 2	FB	4 3			$B + M \rightarrow B$	↑	●	↑	↑	↑	↑
Add Double	ADDD	C3	4 3	D3	5 2	E3	6 2	F3	6 3			$A:B + M:M + 1 \rightarrow A:B$	●	●	↑	↑	↑	↑
Add B to A	ABA									1B	2 1	$A + B \rightarrow A$	↑	●	↑	↑	↑	↑
Add With Carry	ADCA	89	2 2	99	3 2	A9	4 2	B9	4 3			$A + M + C \rightarrow A$	↑	●	↑	↑	↑	↑
	ADCB	C9	2 2	D9	3 2	E9	4 2	F9	4 3			$B + M + C \rightarrow B$	↑	●	↑	↑	↑	↑
AND	ANDA	84	2 2	94	3 2	A4	4 2	B4	4 3			$A \cdot M \rightarrow A$	●	●	↑	↑	R	●
	ANDB	C4	2 2	D4	3 2	E4	4 2	F4	4 3			$B \cdot M \rightarrow B$	●	●	↑	↑	R	●
Bit Test	BIT A	85	2 2	95	3 2	A5	4 2	B5	4 3			$A \cdot M$	●	●	↑	↑	R	●
	BIT B	C5	2 2	D5	3 2	E5	4 2	F5	4 3			$B \cdot M$	●	●	↑	↑	R	●
Clear	CLR					6F	6 2	7F	6 3			$00 \rightarrow M$	●	●	R	S	R	R
	CLRA									4F	2 1	$00 \rightarrow A$	●	●	R	S	R	R
	CLRB									5F	2 1	$00 \rightarrow B$	●	●	R	S	R	R
Compare	CMPA	81	2 2	91	3 2	A1	4 2	B1	4 3			$A - M$	●	●	↑	↑	↑	↑
	CMPB	C1	2 2	D1	3 2	E1	4 2	F1	4 3			$B - M$	●	●	↑	↑	↑	↑
Compare B, A	CBA									11	2 1	$A - B$	●	●	↑	↑	↑	↑
Complement 1's	COM					63	6 2	73	6 3			$\overline{M} \rightarrow M$	●	●	↑	↑	R	S
	COMA									43	2 1	$\overline{A} \rightarrow A$	●	●	↑	↑	R	S
	COMB									53	2 1	$\overline{B} \rightarrow B$	●	●	↑	↑	R	S
Complement 2's (Negate)	NEG					60	6 2	70	6 3			$00 - M \rightarrow M$	●	●	↑	↑	①	②
	NEGA									40	2 1	$00 - A \rightarrow A$	●	●	↑	↑	①	②
	NEGB									50	2 1	$00 - B \rightarrow B$	●	●	↑	↑	①	②
Decimal Adjust, A	DAA									19	2 1	Binary to BCD	●	●	↑	↑	↑	③
Decrement	DEC					6A	6 2	7A	6 3			$M - 1 \rightarrow M$	●	●	↑	↑	④	●
	DECA									4A	2 1	$A - 1 \rightarrow A$	●	●	↑	↑	④	●
	DECB									5A	2 1	$B - 1 \rightarrow B$	●	●	↑	↑	④	●
Exclusive OR	EORA	88	2 2	98	3 2	A8	4 2	B8	4 3			$A \oplus M \rightarrow A$	●	●	↑	↑	R	●
	EORB	C8	2 2	D8	3 2	E8	4 2	F8	4 3			$B \oplus M \rightarrow B$	●	●	↑	↑	R	●
Increment	INC					6C	6 2	7C	6 3			$M + 1 \rightarrow M$	●	●	↑	↑	⑤	●
	INCA									4C	2 1	$A + 1 \rightarrow A$	●	●	↑	↑	⑤	●
	INCB									5C	2 1	$B + 1 \rightarrow B$	●	●	↑	↑	⑤	●
Load Accumulator	LDAA	86	2 2	96	3 2	A6	4 2	B6	4 3			$M \rightarrow A$	●	●	↑	↑	R	●
	LDAB	C6	2 2	D6	3 2	E6	4 2	F6	4 3			$M \rightarrow B$	●	●	↑	↑	R	●
Load Double Accumulator	LDAD	CC	3 3	DC	4 2	EC	5 2	FC	5 3			$M \rightarrow A \quad M + A \rightarrow B$	●	●	↑	↑	R	●

The Condition Code Register notes are listed after Table 7.



TABLE 4 — Continued

ACCUMULATOR AND MEMORY																COND. CODE									
		IMMED.			DIRECT			INDEX			EXTEND			INHERENT			Boolean		5	4	3	2	1	0	
Operations	Mnemonic	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	Arithmetic Operation	H	I	N	Z	V	C		
Multiply Unsigned	MUL													3D	10	1	A X B → A:B	●	●	●	●	●	⑪		
OR, inclusive	ORAA	8A	2	2	9A	3	2	AA	4	2	BA	4	3				A + M → A	●	●	↑	↑	R	●		
	ORAB	CA	2	2	DA	3	2	EA	4	2	FA	4	3				B + M → B	●	●	↑	↑	R	●		
Push Data	PSHA													36	3	1	A Msp, SP - 1 → SP	●	●	●	●	●	●		
	PSHB													37	3	1	B → Msp, SP - 1 → SP	●	●	●	●	●	●		
Pull Data	PULA													32	4	1	SP + 1 → SP, Msp → A	●	●	●	●	●	●		
	PULB													33	4	1	SP + 1 → SP, Msp → B	●	●	●	●	●	●		
Rotate Left	ROL							69	6	2	79	6	3				M	●	●	↑	↑	⑥	↑		
	ROLA													49	2	1	A	●	●	↑	↑	⑥	↑		
	ROLB													59	2	1	B	●	●	↑	↑	⑥	↑		
Rotate Right	ROR							66	6	2	76	6	3				M	●	●	↑	↑	⑥	↑		
	RORA													46	2	1	A	●	●	↑	↑	⑥	↑		
	RORB													56	2	1	B	●	●	↑	↑	⑥	↑		
Shift Left Arithmetic	ASL							68	6	2	78	6	3				M	●	●	↑	↑	⑥	↑		
	ASLA													48	2	1	A	●	●	↑	↑	⑥	↑		
	ASLB													58	2	1	B	●	●	↑	↑	⑥	↑		
Double Shift Left, Arithmetic	ASLD													05	3	1	ACC A/ ACC B	●	●	↑	↑	⑥	↑		
Shift Right Arithmetic	ASR							67	6	2	77	6	3				M	●	●	↑	↑	⑥	↑		
	ASRA													47	2	1	A	●	●	↑	↑	⑥	↑		
	ASRB													57	2	1	B	●	●	↑	↑	⑥	↑		
Shift Right, Logical	LSR							64	6	2	74	6	3				M	●	●	↑	↑	⑥	↑		
	LSRA													44	2	1	A	●	●	↑	↑	⑥	↑		
	LSRB													54	2	1	B			↑	↑		↑		
Double Shift Right Logical	LSRD													04	3	1	ACC A/ ACCB	●	●	R	↑	⑥	↑		
Store Accumulator	STAA				97	3	2	A7	4	2	B7	4	3				A → M	●	●	↑	↑	R	●		
	STAB				D7	3	2	E7	4	2	F7	4	3				B → M	●	●	↑	↑	R	●		
Store Double Accumulator	STAD				DD	4	2	ED	5	2	FD	5	3				A → M B → M + 1	●	●	↑		R	●		
Subtract	SUBA	80	2	2	90	3	2	A0	4	2	B0	4	3				A - M → A	●	●	↑	↑	↑	↑		
	SUBB	C0	2	2	D0	3	2	E0	4	2	F0	4	3				B - M → B	●	●	↑	↑	↑	↑		
Double Subtract	SUBD	83	4	3	93	5	2	A3	6	2	B3	6	3				A:B - M:M + 1 → A:B	●	●	↑	↑	↑	↑		
Subtract Accumulators	SBA													10	2	1	A - B → A	●	●	↑	↑	↑	↑		

The Condition Code Register notes are listed after Table 7.



TABLE 4 — CONTINUED

ACCUMULATOR AND MEMORY		IMMED.			DIRECT			INDEX			EXTEND			INHERENT			Boolean/ Arithmetic Operation		5	4	3	2	1	0
Operations	Mnemonic	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C	
Subtract With Carry	SBCA	82	2	2	92	3	2	A2	4	2	B2	4	3				A - M - C → A	●	●	↑	↑	↑	↑	
	SBCB	C2	2	2	D2	3	2	E2	4	2	F2	4	3				B - M - C → B	●	●	↑	↑	↑	↑	
Transfer Accumulators	TAB													16	2	1	A → B	●	●	↑	↑	R	●	
	TBA													17	2	1	B → A	●	●	↑	↑	R	●	
Text Zero or Minus	TST							6D	6	2	7D	6	3				M - 00	●	●	↑	↑	R	R	
	TSTB													5D	2	1	B - 00	●	●	↑	↑	R	R	

The Condition Code Register notes are listed after Table 7.

ADDED INSTRUCTIONS

In addition to the existing M6800 Instruction Set, the following new instructions are incorporated in the MC6803 Microcomputer.

ABX	Adds the 8-bit unsigned accumulator B to the 16-bit X-Register taking into account the possible carry out of the low order byte of the X-Register.	$IX \leftarrow IX + AC CB$
ADDD	Adds the double precision ACCD* to the double precision value M:M + 1 and places the results in ACCD.	$ACCD \leftarrow (ACCD) + (M:M + 1)$
ASLD	Shifts all bits of ACCD one place to the left. Bit 0 is loaded with zero. The C bit is loaded from the most significant bit of ACCD.	$ACCD \leftarrow 2 \times (ACCD)$
LDD	Loads the contents of double precision memory location into the double accumulator A:B. The condition codes are set according to the data.	$ACCD \leftarrow (M:M + 1)$
LSRD	Shifts all bits of ACCD one place to the right. Bit 15 is loaded with zero. The C bit is loaded from least significant bit to ACCD.	$ACCD \leftarrow (ACCD) \div 2$
MUL	Multiplies the 8 bits in accumulator A with the 8 bits in accumulator B to obtain a 16-bit unsigned number in A:B. ACCA contains MSB of result.	$ACCD \leftarrow ACCA * AC CB$
PSHX	The contents of the index register is pushed onto the stack at the address contained in the stack pointer. The stack pointer is decremented by 2.	$I(IXL), SP \leftarrow (SP) - 1$ $I(IXH), SP \leftarrow (SP) - 1$
PULX	The index register is pulled from the stack beginning at the current address contained in the stack pointer + 1. The stack pointer is incremented by 2 in total.	$SP \leftarrow (SP) + 1; IXH$ $SP \leftarrow (SP) + 1; IXL$
STD	Stores the contents of double accumulator A:B in memory. The contents of ACCD remain unchanged.	$M:M + 1 \leftarrow (ACCD)$
SUBD	Subtracts the contents of M:M + 1 from the contents of double accumulator AB and places the result in ACCD.	$AC CAB \leftarrow (ACCD) - (M:M + 1)$

*ACCD is the 16 bit register (A:B) formed by concatenating the A and B accumulators. The A-accumulator is the most significant byte.



TABLE 5 — INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

Pointer Operations	Mnemonic	IMMED.			DIRECT			INDEX			EXTND			INHERENT			Boolean/ Arithmetic Operation	CONDITION CODE REG.					
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		5	4	3	2	1	0
Compare Index Reg	CPX	8C	4	3	9C	5	2	AC	6	2	BC	6	3				$X_H - M_L X_L - (M + 1)$	●	●	↑	↑	↑	↑
Decrement Index Reg	DEX													09	3	1	$X - 1 \rightarrow X$	●	●	●	↑	●	●
Decrement Stack Pntr	DES													34	3	1	$SP - 1 \rightarrow SP$	●	●	●	●	●	●
Increment Index Reg	INX													08	3	1	$X + 1 \rightarrow X$	●	●	●	↑	●	●
Increment Stack Pntr.	INS													31	3	1	$1 SP + 1 \rightarrow SP$	●	●	●	●	●	●
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	5	2	FE	5	3				$M \rightarrow X_H (M + 1) \rightarrow X_L$	●	●	⑦	↑	R	●
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	5	2	BE	5	3				$M \rightarrow SP_H (M + 1) \rightarrow SP_L$	●	●	⑦	↑	R	●
Store Index Reg	STX				DF	4	2	EF	5	2	FF	5	3				$X_H \rightarrow M_L X_L \rightarrow (M + 1)$	●	●	⑦	↑	R	●
Store Stack Pntr	STS				9F	4	2	AF	5	2	5	3	3				$SP_H \rightarrow M_L SP_L \rightarrow (M + 1)$	●	●	⑦	↑	R	●
Index Reg \rightarrow Stack Pntr	TXS													35	3	1	$X - 1 \rightarrow SP$	●	●	●	●	●	●
Stack Pntr \rightarrow Index Reg	TSX													30	3	1	$SP + 1 \rightarrow X$	●	●	●	●	●	●
ADD	ABX													3A	3	1	$B + X \rightarrow X$	●	●	●	●	●	●
Push Data	PSHX													3C	4	1	$X_L \rightarrow M_{SP} SP - 1 \rightarrow SP$ $X_H \rightarrow H_{SP} SP - 1 \rightarrow SP$	●	●	●	●	●	●
Pull Data	PULX													38	5	1	$SP + 1 \rightarrow SP M_{SP} \rightarrow X_H$ $SP + 1 \rightarrow SP M_{SP} \rightarrow X_L$	●	●	●	●	●	●

The Condition Code Register notes are listed after Table 7.



TABLE 6 — JUMP AND BRANCH INSTRUCTIONS


		RELATIVE			INDEX			EXTEND			INHERENT			DIRECT			5 4 3 2 1 0						
Operations	Mnemonic	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	Branch Test	H	I	N	Z	V	C
Branch Always	BRA	20	3	2													None	●	●	●	●	●	●
Branch If Carry Clear	BCC	24	3	2													C = 0	●	●	●	●	●	●
Branch If Carry Set	BCS	25	3	2													C = 1	●	●	●	●	●	●
Branch If = Zero	BEO	27	3	2													Z = 1	●	●	●	●	●	●
Branch If ≥ Zero	BGE	2C	3	2													N ⊕ V = 0	●	●	●	●	●	●
Branch If > Zero	BGT	2E	3	2													Z + (N ⊕ V) = 0	●	●	●	●	●	●
Branch If Higher	BHI	22	3	2													C + Z = 0	●	●	●	●	●	●
Branch If ≤ Zero	BLE	2F	3	2													Z + (N ⊕ V) =	●	●	●	●	●	●
Branch If Lower Or Same	BLS	23	3	2													C + Z =	●	●	●	●	●	●
Branch If < Zero	BLT	2D	3	2													N ⊕ V = 1	●	●	●	●	●	●
Branch If Minus	BMI	2B	3	2													N = 1	●	●	●	●	●	●
Branch If Not Equal Zero	BNE	26	3	2													Z = 0	●	●	●	●	●	●
Branch If Overflow Clear	BVC	28	3	2													V = 0	●	●	●	●	●	●
Branch If Overflow Set	BVS	29	2	2													V = 1	●	●	●	●	●	●
Branch If Plus	BPL	2A	3	2													N = 0	●	●	●	●	●	●
Branch To Subroutine	BSR	8D	6	2														●	●	●	●	●	●
Jump	JMP				6E	3	2	7E	3	3							See Special Operations	●	●	●	●	●	●
Jump To Subroutine	JSR				AD	6	2	BD	6	3				9D	5	2		●	●	●	●	●	●
No Operation	NOP										01	2	1				Advances Prog.Cntr. Only	●	●	●	●	●	●
Return From Interrupt	RTI										3B	10	1					← ⑧ →					
Return From Subroutine	RTS										39	5	1					●	●	●	●	●	●
Software Interrupt	SWI										3F	2	1					●	●	●	●	●	●
Wait For Interrupt*	WAI										3E	9	1				See Special Operatios	●	⑨	●	●	●	●

The Condition Code Register notes are listed after Table 7.

*WAI puts R/W high; Address Bus goes to FFFF; Data Bus goes to the three-state level.



TABLE 7 — CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

					COND. CODE REG.						
INHERENT					5 4 3 2 1 0						
Operations	Mnemonic	OP	~	#	Boolean Operation	H	I	N	Z	V	C
Clear Carry	CLC	OC	2	1	$0 \rightarrow C$	●	●	●	●	●	R
Clear Interrupt Mask	CLI	OE	2	1	$0 \rightarrow I$	●	R	●	●	●	●
Clear Overflow	CLV	OA	2	1	$0 \rightarrow V$	●	●	●	●	R	●
Set Carry	SEC	OD	2	1	$1 \rightarrow C$	●	●	●	●	●	S
Set Interrupt Mask	SEI	OF	2	1	$1 \rightarrow I$	●	S	●	●	●	●
Set Overflow	SEV	OB	2	1	$1 \rightarrow V$	●	●	●	●	S	●
Accumulator A \rightarrow CCR	TAP	06	2	1	$A \rightarrow CCR$						
CCR \rightarrow Accumulator A	TPA	07	2	1	$CCR \rightarrow A$	●	●	●	●	●	●

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- 1 (Bit V) Test: Result = 10000000?
- 2 (Bit C) Test: Result = 00000000?
- 3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set).
- 4 (Bit V) Test: Operand = 10000000 prior to execution?
- 5 (Bit V) Test: Operand = 01111111 prior to execution?
- 6 (Bit V) Test: Set equal to result of $N \oplus C$ after shift has occurred.
- 7 (Bit N) Test: Result less than zero? (Bit 15 = 1)
- 8 (All) Load Condition Code Register from Stack
(See Special Operation)
- 9 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
- 10 (All) Set according to the contents of Accumulator A
- 11 C = R7

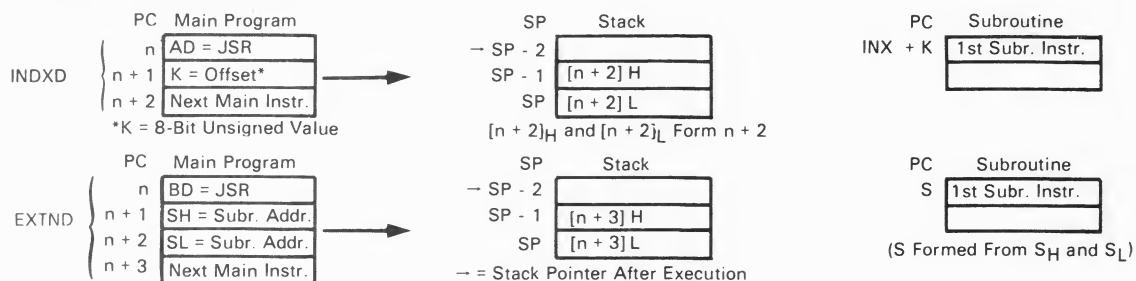
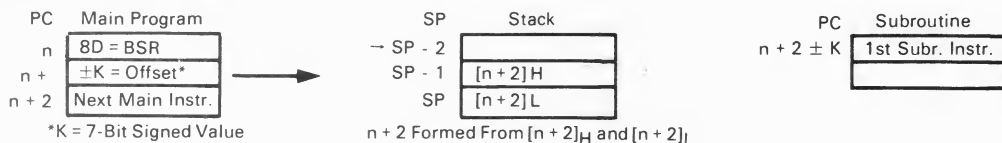
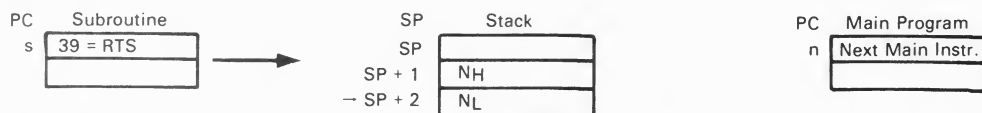
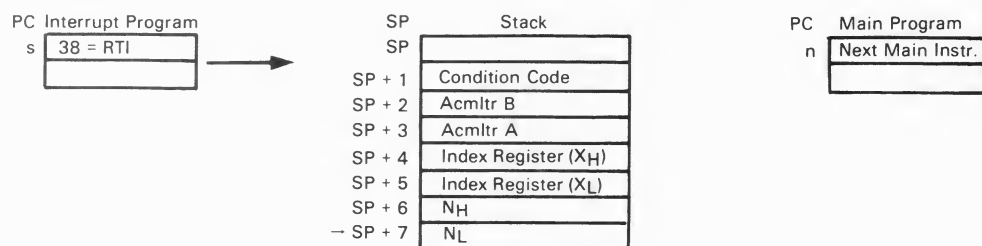


TABLE 8 — INSTRUCTION EXECUTION TIMES IN
MACHINE CYCLES

	ACCX	IMMEDIATE	DIRECT	EXTENDED	INDEXED	INHERENT	RELATIVE		ACCX	IMMEDIATE	DIRECT	EXTENDED	INDEXED	INHERENT	RELATIVE
ABA	●	●	●	●	●	2	●	INX	●	●	●	●	●	3	●
ABX	●	●	●	●	●	3	●	JMP	●	●	●	3	3	●	●
ADC	●	2	3	4	4	●	●	JSR	●	●	5	6	6	●	●
ADD	●	2	3	4	4	●	●	LDA	●	2	3	4	4	●	●
ADDD	●	4	5	6	6	●	●	LDD	●	3	4	5	5	●	●
AND	●	2	3	4	4	●	●	LDS	●	3	4	5	5	●	●
ASL	2	●	●	6	6	●	●	LDX	●	3	4	5	5	●	●
ASLD	●	●	●	●	●	3	●	LSR	2	●	●	6	6	●	●
ASR	2	●	●	6	6	●	●	LSRD	●	●	●	●	●	3	●
BCC	●	●	●	●	●	●	3	MUL	●	●	●	●	●	10	●
BCS	●	●	●	●	●	●	3	NEG	2	●	●	6	6	●	●
BEQ	●	●	●	●	●	●	3	NOP	●	●	●	●	●	2	●
BGE	●	●	●	●	●	●	3	ORA	●	2	3	4	4	●	●
BGT	●	●	●	●	●	●	3	PSH	3	●	●	●	●	●	●
BHI	●	●	●	●	●	●	3	PSHX	●	●	●	●	●	4	●
BIT	●	2	3	4	4	●	●	PUL	4	●	●	●	●	●	●
BLE	●	●	●	●	●	●	3	PULX	●	●	●	●	●	5	●
BLS	●	●	●	●	●	●	3	ROL	2	●	●	6	6	●	●
BLT	●	●	●	●	●	●	3	ROR	2	●	●	6	6	●	●
BMI	●	●	●	●	●	●	3	RTI	●	●	●	●	●	10	●
BNE	●	●	●	●	●	●	3	RTS	●	●	●	●	●	5	●
BPL	●	●	●	●	●	●	3	SBA	●	●	●	●	●	2	●
BRA	●	●	●	●	●	●	3	SBC	●	2	3	4	4	●	●
BSR	●	●	●	●	●	●	6	SEC	●	●	●	●	●	2	●
BVC	●	●	●	●	●	●	3	SEI	●	●	●	●	●	2	●
BVS	●	●	●	●	●	●	3	SEV	●	●	●	●	●	2	●
CBA	●	●	●	●	●	2	●	STA	●	●	3	4	4	●	●
CLC	●	●	●	●	●	2	●	STD	●	●	4	5	5	●	●
CLI	●	●	●	●	●	2	●	STS	●	●	4	5	5	●	●
CLR	2	●	●	6	6	●	●	STX	●	●	4	5	5	●	●
CLV	●	●	●	●	●	2	●	SUB	●	2	3	4	●	●	●
CMP	●	2	3	4	4	●	●	SUBD	●	4	5	6	6	●	●
COM	2	●	●	6	6	●	●	SWI	●	●	●	●	●	12	●
CPX	●	4	5	6	6	●	●	TAB	●	●	●	●	●	2	●
DAA	●	●	●	●	●	2	●	TAP	●	●	●	●	●	2	●
DEC	2	●	●	6	6	●	●	TBA	●	●	●	●	●	2	●
DES	●	●	●	●	●	3	●	TPA	●	●	●	●	●	2	●
DEX	●	●	●	●	●	3	●	TST	2	●	●	6	6	●	●
EOR	●	2	3	4	4	●	●	TSX	●	●	●	●	●	3	●
INC	2	●	●	6	6	●	●	TXS	●	●	●	●	●	3	●
INS	●	●	●	●	●	3	●	WAI	●	●	●	●	●	9	●



FIGURE 21 — SPECIAL OPERATIONS

JSR, JUMP TO SUBROUTINE:**BSR, BRANCH TO SUBROUTINE:****JMP, JUMP:****RTS, RETURN FROM SUBROUTINE:****RTI, RETURN FROM INTERRUPT:**

SUMMARY OF CYCLE BY CYCLE OPERATION

Table 9 provides a detailed description of the information present on the Address Bus, Data Bus, and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expanded results during debug of both software and hardware as the control program is executed. The information is categorized in groups according to addressing mode and number of cycles per instruction. (In general, instructions with the same addressing mode and number of cycles execute in the same manner; exception indicated in the table.)

TABLE 9 — CYCLE BY CYCLE OPERATION

ADDRESS MODE & INSTRUCTIONS	CYCLES	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
IMMEDIATE					
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1 2	Op Code Address Op Code Address + 1	1 1	Op Code Operand Data
LDS LDX	3	1 2 3	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte)
CPX SUBD ADDD	4	1 2 3 4	Op Code Address Op Code Address + 1 Op Code Address + 2 Address Bus FFFF	1 1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte) Low Byte of Restart Vector
DIRECT					
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1 2 3	Op Code Address Op Code Address + 1 Address of Operand	1 1 1	Op Code Address of Operand Operand Data
STA	3	1 2 3	Op Code Address Op Code Address + 1 Destination Address	1 1 0	Op Code Destination Address Data from Accumulator
LDS LDX LDD	4	1 2 3 4	Op Code Address Op Code Address + 1 Address of Operand Operand Address + 1	1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte)
STS STX STD	4	1 2 3 4	Op Code Address Op Code Address + 1 Address of Operand Address of Operand + 1	1 1 0 0	Op Code Address of Operand Register Data (High Order Byte) Register Data (Low Order Byte)
CPX SUBD ADDD	5	1 2 3 4 5	Op Code Address Op Code Address + 1 Operand Address Operand Address + 1 Address Bus FFFF	1 1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte) Low Byte of Restart Vector
JSR	5	1 2 3 4 5	Op Code Address Op Code Address + 1 Subroutine Address Stack Pointer Stack Pointer + 5	1 1 1 0 0	Op Code Irrelevant Data First Subroutine Op Code Return Address (Low Order Byte) Return Address (High Order Byte)

(Continued)



TABLE 9 — CYCLE BY CYCLE OPERATION
(cont.)

ADDRESS MODE & INSTRUCTIONS	CYCLE	CYCLE #	ADDRESS BUS	R/ \overline{W} LINE	DATA BUS
INDEXED					
JMP	3	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	1	Operand Data
STA	4	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	0	Operand Data
LDS LDX LDD	5	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	1	Operand Data (High Order Byte)
		5	Index Register Plus Offset + 1	1	Operand Data (Low Order Byte)
STS STX STD	5	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	0	Operand Data (High Order Byte)
		5	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
ASL LSR ASR NEG CLR ROL COM ROR DEC TST (1) INC	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register Plus Offset	1	Current Operand Data
		5	Address Bus FFFF	1	Low Byte of Restart Vector
		6	Index Register Plus Offset	0	New Operand Data
CPX SUBD ADDD	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register + Offset	1	Operand Data (High Order Byte)
		5	Index Register + Offset + 1	1	Operand Data (Low Order Byte)
		6	Address Bus FFFF	1	Low Byte of Restart Vector
JSR	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Index Register + Offset	1	First Subroutine Op Code
		5	Stack Pointer	0	Return Address (Low Order Byte)
		6	Stack Pointer - 1	0	Return Address (High Order Byte)

(Continued)



TABLE 9 — CYCLE BY CYCLE OPERATION
(cont.)

ADDRESS MODE & INSTRUCTIONS	CYCLE	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
EXTENDED					
JMP	3	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Jump Address (High Order Byte)
		3	Op Code Address + 2	1	Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Address of Operand
		3	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	Address of Operand	1	Operand Data
STA A STA B	4	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Destination Address (High Order Byte)
		3	Op Code Address + 2	1	Destination Address (Low Order Byte)
		4	Operand Destination Address	0	Data from Accumulator
LDS LDX LDD	5	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	Address of Operand	1	Operand Data (High Order Byte)
		5	Address of Operand + 1	1	Operand Data (Low Order Byte)
STS STX STD	5	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	Address of Operand	0	Operand Data (High Order Byte)
		5	Address of Operand + 1	0	Operand Data (Low Order Byte)
ASL LSR ASR NEG CLR ROL COM ROR DEC TST (1) INC	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	Address of Operand	1	Current Operand Data
		5	Address Bus FFFF	1	Low Byte of Restart Vector
		6	Address of Operand	0	New Operand Data
CPX SUBD ADDD	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Operand Address (High Order Byte)
		3	Op Code Address + 2	1	Operand Address (Low Order Byte)
		4	Operand Address	1	Operand Data (High Order Byte)
		5	Operand Address + 1	1	Operand Data (Low Order Byte)
		6	Address Bus FFFF	1	Low Byte of Restart Vector
JSR	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Address of Subroutine (High Order Byte)
		3	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
		4	Subroutine Starting Address	1	Op Code of Next Instruction
		5	Stack Pointer	0	Return Address (Low Order Byte)
		6	Stack Pointer - 1	0	Address of Operand (High Order Byte)

(Continued)



TABLE 9 — CYCLE BY CYCLE OPERATION
(cont.)

ADDRESS MODE & INSTRUCTIONS	CYCLES	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
INHERENT					
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1 2	Op Code Address Op Code Address + 1	1 1	Op Code Op Code of Next Instruction
ABX	3	1 2 3	Op Code Address Op Code Address + 1 Address Bus FFFF	1 1 1	Op Code Irrelevant Data Low Byte of Restart Vector
ASLD LSRD	3	1 2 3	Op Code Address Op Code Address + 1 Address Bus FFFF	1 1 1	Op Code Irrelevant Data Low Byte of Restart Vector
DES INS	3	1 2 3	Op Code Address Op Code Address + 1 Previous Register Contents	1 1 1	Op Code Op Code of Next Instruction Irrelevant Data
INX DEX	3	1 2 3	Op Code Address Op Code Address + 1 Address Bus FFFF	1 1 1	Op Code Op Code of Next Instruction Low Byte of Restart Vector
PSHA PSHB	3	1 2 3	Op Code Address Op Code Address + 1 Stack Pointer	1 1 0	Op Code Op Code of Next Instruction Accumulator Data
ISX	3	1 2 3	Op Code Address Op Code Address + 1 Stack Pointer	1 1 1	Op Code Op Code of Next Instruction Irrelevant Data
TXS	3	1 2 3	Op Code Address Op Code Address + 1 Address Bus FFFF	1 1 1	Op Code Op Code of Next Instruction Low Byte of Restart Vector
PULA PULB	4	1 2 3 4	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer + 1	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data
PSHX	4	1 2 3 4	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer - 1	1 1 0 0	Op Code Irrelevant Data Index Register (Low Order Byte) Index Register (High Order Byte)
PULX	5	1 2 3 4 5	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer + 1 Stack Pointer + 2	1 1 1 1 1	Op Code Irrelevant Data Irrelevant Data Index Register (High Order Byte) Index Register (Low Order Byte)

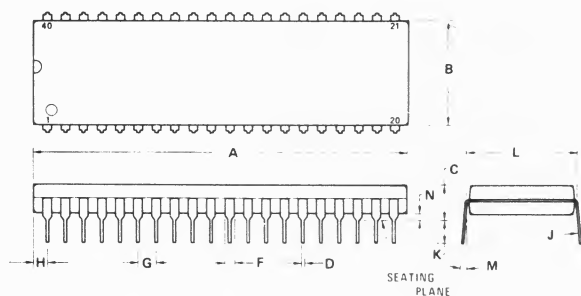
(Continued)



TABLE 9 — CYCLE BY CYCLE OPERATION
(cont.)

ADDRESS MODE & INSTRUCTIONS	CYCLES	CYCLE #	ADDRESS BUS	R/W LINE	DATA BUS
INHERENT (cont.)					
WAI	9	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Op Code of Next Instruction
		3	Stack Pointer	0	Return Address (Low Order Byte)
		4	Stack Pointer - 1	0	Return Address (High Order Byte)
		5	Stack Pointer - 2	0	Index Register (Low Order Byte)
		6	Stack Pointer - 3	0	Index Register (High Order Byte)
		7	Stack Pointer - 4	0	Contents of Accumulator A
		8	Stack Pointer - 5	0	Contents of Accumulator B
		9	Stack Pointer - 6	1	Contents of Cond. Code Register.
RTI	10	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Irrelevant Data
		3	Stack Pointer	1	Irrelevant Data
		4	Stack Pointer + 1	1	Contents of Cond. Code Register from Stack
		5	Stack Pointer + 2	1	Contents of Accumulator B from Stack
		6	Stack Pointer + 3	1	Contents of Accumulator A from Stack
		7	Stack Pointer + 4	1	Index Register from Stack (High Order Byte)
		8	Stack Pointer + 5	1	Index Register from Stack (Low Order Byte)
		9	Stack Pointer + 6	1	Next Instruction Address from Stack (High Order Byte)
		10	Stack Pointer + 7	1	Next Instruction Address from Stack (Low Order Byte)
SWI	12	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Irrelevant Data
		3	Stack Pointer	0	Return Address (Low Order Byte)
		4	Stack Pointer - 1	0	Return Address (High Order Byte)
		5	Stack Pointer - 2	0	Index Register (Low Order Byte)
		6	Stack Pointer - 3	0	Index Register (High Order Byte)
		7	Stack Pointer - 4	0	Contents of Accumulator A
		8	Stack Pointer - 5	0	Contents of Accumulator B
		9	Stack Pointer - 6	0	Contents of Cond. Code Register
		10	Stack Pointer - 7	1	Irrelevant Data
		11	Vector Address FFFA (Hex)	1	Address of Subroutine (High Order Byte)
		12	Vector Address FFFB (Hex)	1	Address of Subroutine (Low Order Byte)
BCC BHT BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMT BVS	3	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Branch Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
BSR	6	1	Op Code Address	1	Op Code
		2	Op Code Address + 1	1	Branch Offset
		3	Address Bus FFFF	1	Low Byte of Restart Vector
		4	Subroutine Starting Address	1	Op Code of Next Instruction
		5	Stack Pointer	0	Return Address (Low Order Byte)
		6	Stack Pointer - 1	0	Return Address (High Order Byte)



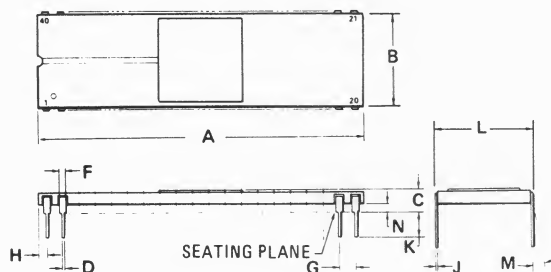


DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.69	52.45	2.035	2.065
B	13.72	14.22	0.540	0.560
C	3.94	5.08	0.155	0.200
D	0.36	0.56	0.014	0.022
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.38	0.008	0.015
K	2.92	3.43	0.115	0.135
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.02	0.020	0.040

CASE 711-03

NOTES:

1. POSITIONAL TOLERANCE OF LEADS (D), SHALL BE WITHIN 0.25 mm (0.010) AT MAXIMUM MATERIAL CONDITION, IN RELATION TO SEATING PLANE AND EACH OTHER.
2. DIMENSION L TO CENTER OF LEADS WHEN FORMED PARALLEL.
3. DIMENSION B DOES NOT INCLUDE MOLD FLASH.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.94	15.34	0.588	0.604
C	3.05	4.06	0.120	0.160
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.99	15.49	0.590	0.610
M	—	10°	—	10°
N	1.02	1.52	0.040	0.060

CASE 715-03

NOTES:

1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX MAT'L CONDITION.
2. DIMENSION "L" TO CENTER OF LEADS WHEN FORMED PARALLEL.

Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.


MOTOROLA Semiconductor Products Inc.

**MOTOROLA**

SEMICONDUCTORS

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

Advance Information

8-BIT MICROCOMPUTER UNIT

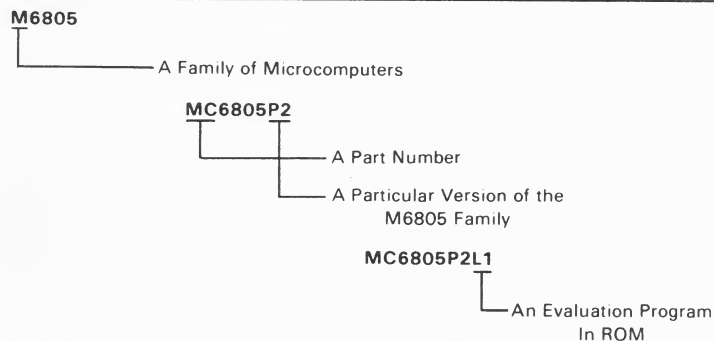
The MC6805P2 Microcomputer Unit (MCU) is a member of the M6805 Family of microcomputers. This 8-bit microcomputer contains a CPU, on-chip clock, ROM, RAM, I/O and timer. It is designed for the user who needs an economical microcomputer with the proven capabilities of the M6800-based instruction set. Table 9 compares the key features of the M6805 Family of microcomputers. The following are some of the hardware and software highlights of the MCU.

HARDWARE FEATURES:

- 8-Bit Architecture
- 64 Bytes of RAM
- Memory Mapped I/O
- 1100 Bytes of User ROM
- Internal 8-Bit Timer with 7-Bit Prescaler
- Vectored Interrupts — External and Timer
- 20 TTL/CMOS Compatible I/O Lines;
8 Lines LED Compatible
- On-Chip Clock Circuit
- Self-Check Mode
- Master Reset
- Low Voltage Inhibit
- Complete Development System Support on EXORciser
- 5 Vdc Single Supply

SOFTWARE FEATURES:

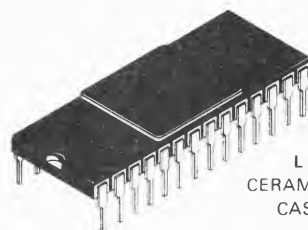
- Similar to M6800
- Byte Efficient Instruction Set
- Easy to Program
- True Bit Manipulation
- Bit Test and Branch Instructions
- Versatile Interrupt Handling
- Powerful Indexed Addressing for Tables
- Full Set of Conditional Branches
- Memory Usable as Registers/Flags
- Single Instruction Memory Examine/Change
- 10 Powerful Addressing Modes
- All Addressing Modes Apply to ROM, RAM and I/O

**MC6805P2**

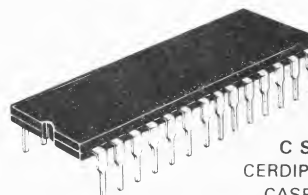
HMOS

(HIGH DENSITY
N-CHANNEL, SILICON-GATE
DEPLETION LOAD)

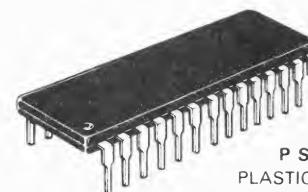
8-BIT MICROCOMPUTER



L SUFFIX
CERAMIC PACKAGE
CASE 719-03



C SUFFIX
CERDIP PACKAGE
CASE 733-02



P SUFFIX
PLASTIC PACKAGE
CASE 710-02

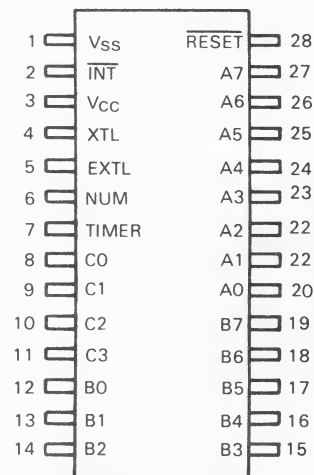
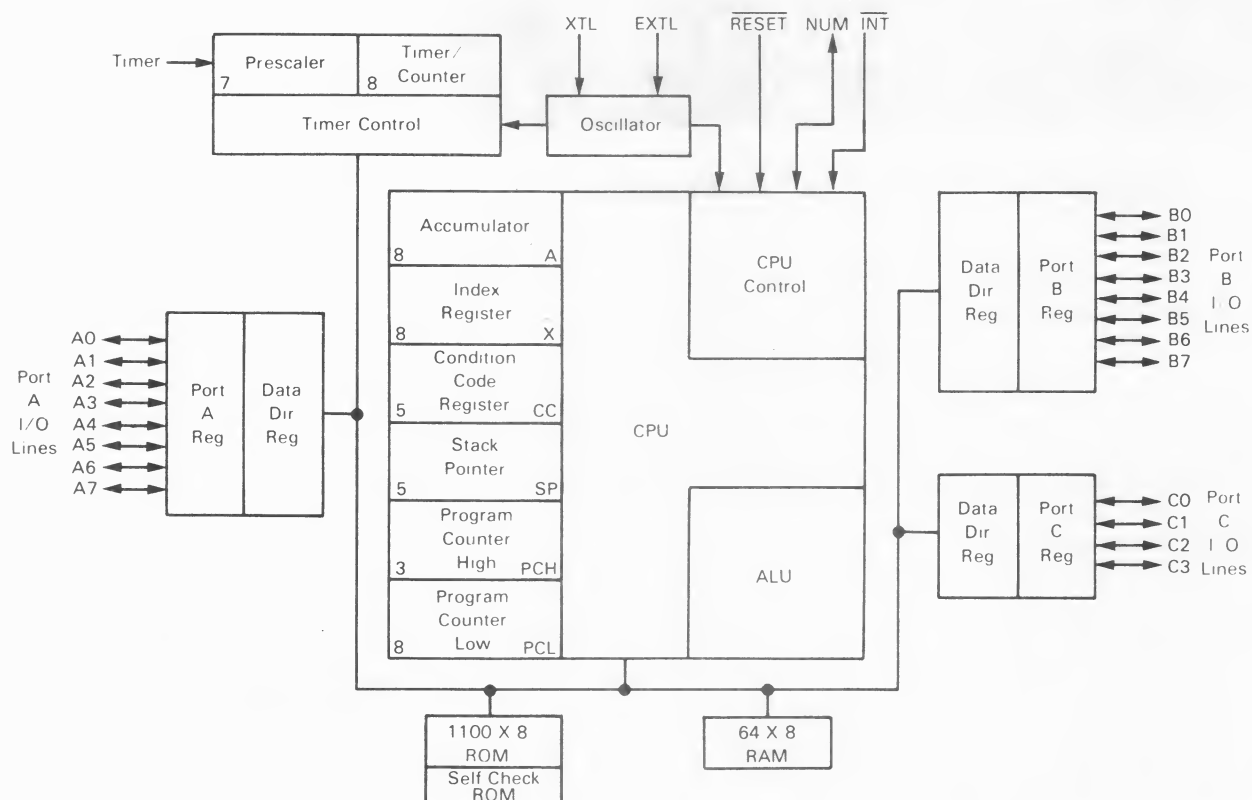
FIGURE 1 — PIN ASSIGNMENTS

FIGURE 2 — MC6805P2 HMOS MICROCOMPUTER BLOCK DIAGRAM



MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	Vdc
Input Voltage	V_{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to 70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C
Thermal Resistance	θ_{JA}	85	°C/W
		50	
		51	

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields, however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit. For proper operation it is recommended that V_{in} and V_{out} be constrained to the range $V_{SS} - (V_{in} \text{ or } V_{out}) + V_{CC}$.

ELECTRICAL CHARACTERISTICS ($V_{CC} = +5.25 \text{ Vdc} \pm V_{dc}$, $V_{SS} = \text{GND}$, $T_A = 0 - 70^\circ \text{C}$ unless otherwise noted)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage	$\overline{\text{RESET}}$	V_{IH}	4.0	—	V_{CC}
	$\overline{\text{INT}}$	V_{IH}	—	2.2	V_{CC}
	All Other	V_{IH}	$V_{SS} + 2.0$	—	V_{CC}
Input High Voltage Timer	Timer Mode	V_{IH}	$V_{SS} + 2.0$	—	V_{CC}
	Self-Check Mode	V_{IH}	—	9.0	15.0
Input Low Voltage	$\overline{\text{RESET}}$	V_{IL}	$V_{SS} - 0.3$	0.8	V_{CC}
	$\overline{\text{INT}}$	V_{IL}	—	2.0	V_{CC}
	All Other	V_{IL}	$V_{SS} - 0.3$	—	$V_{SS} + 0.8$
$\overline{\text{INT}}$ Hysteresis	V_H	—	100	—	mVdc
Power Dissipation	P_D	—	350	—	mW
Input Capacitance	EXTL	C_{in}	—	20	pF
	All Other	C_{in}	—	10	pF
Low Voltage Recover	LVR	—	—	4.75	Vdc
Low Voltage Inhibit	LVI	—	4.5	—	

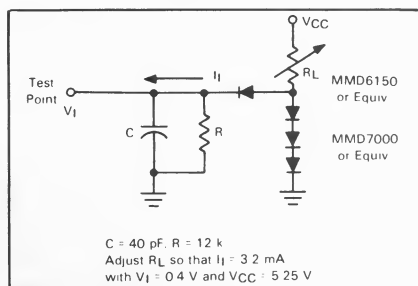
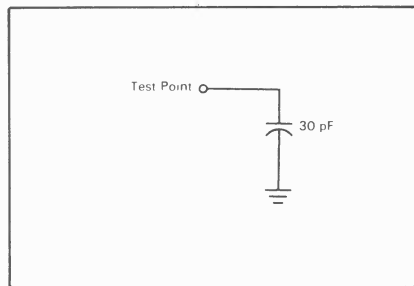
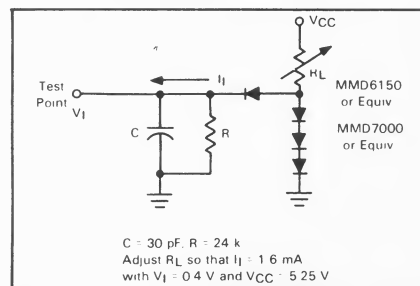


SWITCHING CHARACTERISTICS ($V_{CC} = +5.25 \text{ V} \pm 0.5 \text{ Vdc}$, $V_{SS} = \text{GND}$, $T_A = 0^\circ - 70^\circ\text{C}$ unless otherwise noted)

Characteristic	Symbol	Min	Typ	Max	Unit
Clock Frequency	f_{cl}	0.4	—	4.0	MHz
Cycle Time	t_{CYC}	1.0	—	10	μs
INT Pulse Width	t_{IWL}	$t_{CYC} + 250$	—	—	ns
RESET Pulse Width	t_{RWL}	$t_{CYC} + 250$	—	—	ns
Delay Time Reset (External Cap. = $0.47 \mu\text{F}$)	t_{RHL}	20	50	—	ms

PORT ELECTRICAL CHARACTERISTICS ($V_{CC} = +5.25 \text{ V} \pm 0.5 \text{ Vdc}$, $V_{SS} = \text{GND}$, $T_A = 0^\circ$ to 70°C unless otherwise noted).

Characteristic	Symbol	Min	Typ	Max	Unit
Port A					
Output Low Voltage $I_{load} = 1.6 \text{ mAdc}$	V_{OL}	—	—	0.4	Vdc
Output High Voltage $I_{load} = -100 \mu\text{Adc}$	V_{OH}	2.4	—	—	Vdc
Output High Voltage $I_{load} = -10 \mu\text{Adc}$	V_{OH}	3.5	—	—	Vdc
Input High Voltage $I_{load} = -300 \mu\text{Adc}$ (max)	V_{IH}	$V_{SS} + 2.0$	—	V_{CC}	Vdc
Input Low Voltage $I_{load} = -500 \mu\text{Adc}$ (max)	V_{IL}	$V_{SS} - 0.3$	—	$V_{SS} + 0.8$	Vdc
Port B					
Output Low Voltage $I_{load} = 3.2 \text{ mAdc}$	V_{OL}	—	—	0.4	Vdc
Output Low Voltage $I_{load} = 10 \text{ mAdc}$ (sink)	V_{OL}	—	—	1.0	Vdc
Output High Voltage $I_{load} = -200 \mu\text{Adc}$	V_{OH}	2.4	—	—	Vdc
Darlington Current Drive (Source) $V_O = 1.5 \text{ Vdc}$	I_{OH}	-1.0	—	-10	mAdc
Input High Voltage	V_{IH}	$V_{SS} + 2.0$	—	V_{CC}	Vdc
Input Low Voltage	V_{IL}	$V_{SS} - 0.3$	—	$V_{SS} + 0.8$	Vdc
Port C					
Output Low Voltage $I_{load} = 1.6 \text{ mAdc}$	V_{OL}	—	—	0.4	Vdc
Output High Voltage $I_{load} = -100 \mu\text{Adc}$	V_{OH}	2.4	—	—	Vdc
Input High Voltage	V_{IH}	$V_{SS} + 2.0$	—	V_{CC}	Vdc
Input Low Voltage	V_{IL}	$V_{SS} - 0.3$	—	$V_{SS} + 0.8$	Vdc
Off-State Input Current					
Three-State Ports B & C	I_{TSI}	—	2	20	μAdc
Input Current					
Timer at $V_{in} = (0.4 \text{ to } 2.4 \text{ Vdc})$	I_{in}	—	—	20	μAdc

FIGURE 3 — TTL EQUIV. TEST LOAD (PORT B)**FIGURE 4 — CMOS EQUIV. TEST LOAD (PORT A)****FIGURE 5 — TTL EQUIV. TEST LOAD (PORTS A AND C)**

SIGNAL DESCRIPTION

The input and output signals for the MCU shown in Figure 1 are described in the following paragraphs.

VCC AND VSS — Power is supplied to the MCU using these two pins. VCC is +5.25 Vdc ± 0.5 V. VSS is the ground connection.

INT — This pin provides the capability for applying an external interrupt to the MCU. Refer to **INTERRUPTS** for additional information.

XTAL AND EXTAL — These pins provide control input for the on-chip clock circuit. A crystal (AT cut, 4 MHz maximum) or a resistor can be connected to these pins to provide the internal oscillator with varying degrees of stability. Refer to **INTERNAL OSCILLATOR OPTIONS** for recommendations about these inputs.

TIMER — This pin allows an external input to be used to decrement the internal timer circuitry. Refer to **TIMER** for additional information about the timer circuitry.

RESET — This pin allows resetting of the MCU at times other than the automatic resetting capability already in the MCU. Refer to **RESETS** for additional information.

NUM — This pin is not for user application and should be connected to ground.

INPUT/OUTPUT LINES (A0-A7, B0-B7, C0-C3)

These 20 lines are arranged into two 8-bit ports (A and B) and one 4-bit port (C). All lines are programmable as either inputs or outputs under software control of the data direction registers. Refer to **INPUTS/OUTPUTS** for additional information.

MEMORY

The MCU memory is configured as shown in Figure 6. During the processing of an interrupt, the contents of the MCU registers are pushed onto the stack in the order shown in Figure 7. Since the stack pointer decrements during pushes, the low order byte (PCL) of the program counter is stacked first; then the high order three bits (PCH) are stacked. This ensures that the program counter is loaded correctly as the stack pointer increments when it pulls data from the stack. A subroutine call will cause only the program counter (PCH, PCL) contents to be pushed onto the stack.

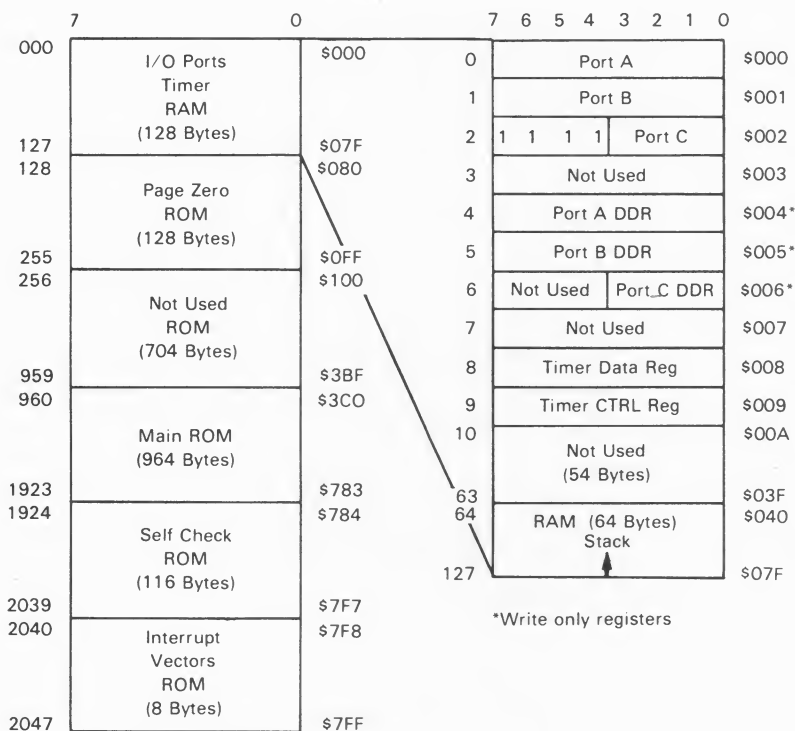
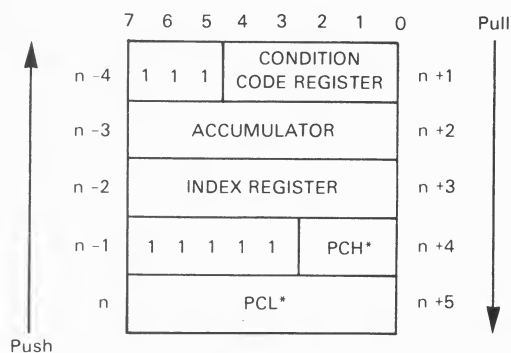
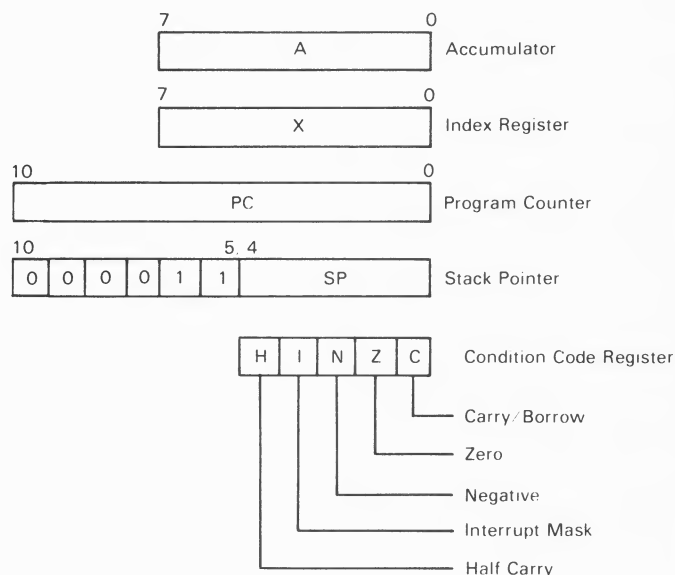
FIGURE 6 — MCU MEMORY CONFIGURATION

FIGURE 7 — INTERRUPT STACKING ORDER



* For subroutine calls, only PCH and PCL are stacked

FIGURE 8 — PROGRAMMING MODEL



REGISTERS

The MCU has five registers available to the programmer. They are shown in Figure 8 and are explained in the following paragraphs.

ACCUMULATOR (A) — The accumulator is a general purpose 8-bit register used to hold operands and results of arithmetic calculations or data manipulations.

INDEX REGISTER (X) — The index register is an 8-bit register used for the indexed addressing mode. It contains an 8-bit address that may be added to an offset value to create an effective address. The index register can also be used for limited calculations and data manipulations when using read/modify write instructions. When not required by a code sequence being executed, the index register can be used as a temporary storage area.

PROGRAM COUNTER (PC) — The program counter is an 11-bit register that contains the address of the next instruction to be executed.

STACK POINTER (SP) — The stack pointer is an 11-bit register that contains the address of the next free location on the stack. Initially, the stack pointer is set to location \$07F and is decremented as data is being pushed onto the stack and incremented as data is being pulled from the stack. The six most significant bits of the stack pointer are permanently set to 000011. During a MCU reset or the reset stack pointer (RSP) instruction, the stack pointer is set to location \$07F. Subroutines and interrupts may be nested down to location \$061 which allows the programmer to use up to 15 levels of subroutine calls.

CONDITION CODE REGISTER (CC) — The condition code register is a 5-bit register in which each bit is used to indicate or flag the results of the instruction just executed. These bits can be individually tested by a program and specific action taken as a result of their state. Each individual condition code register bit is explained in the following paragraphs.

Half Carry (H) — Used during arithmetic operations (ADD and ADC) to indicate that a carry occurred between bits 3 and 4.

Interrupt (I) — This bit is set to mask the timer and external interrupt (\overline{INT}). If an interrupt occurs while this bit is set it is latched and will be processed as soon as the interrupt bit is reset.

Negative (N) — Used to indicate that the result of the last arithmetic, logical or data manipulation was negative (bit 7 in result equal to a logical one).

Zero (Z) — Used to indicate that the result of the last arithmetic, logical or data manipulation was zero.

Carry/Borrow (C) — Used to indicate that a carry or borrow out of the arithmetic logic unit (ALU) occurred during the last arithmetic operation. This bit is also affected during bit test and branch instructions, shifts, and rotates.

TIMER

The MCU timer circuitry is shown in Figure 9. The 8-bit counter is loaded under program control and counts down toward zero as soon as the clock input is applied. When the timer reaches zero the timer interrupt request bit (bit 7) in the timer control register is set. The MCU responds to this interrupt by saving the present MCU state in the stack, fetching the timer interrupt vector from locations \$7F8 and \$7F9 and executing the interrupt routine. The timer interrupt can be masked by setting the timer interrupt mask bit (bit 6) in the timer control register. The interrupt bit (I bit) in the condition code register will also prevent a timer interrupt from being processed.

The clock input to the timer can be from an external source applied to the TIMER input pin or it can be the internal $\phi 2$ signal. Note that when the $\phi 2$ signal is used as the source it can be gated by an input applied to the TIMER input pin allowing the user to easily perform pulse-width measurements. The source of the clock input is one



FIGURE 9 — TIMER BLOCK DIAGRAM

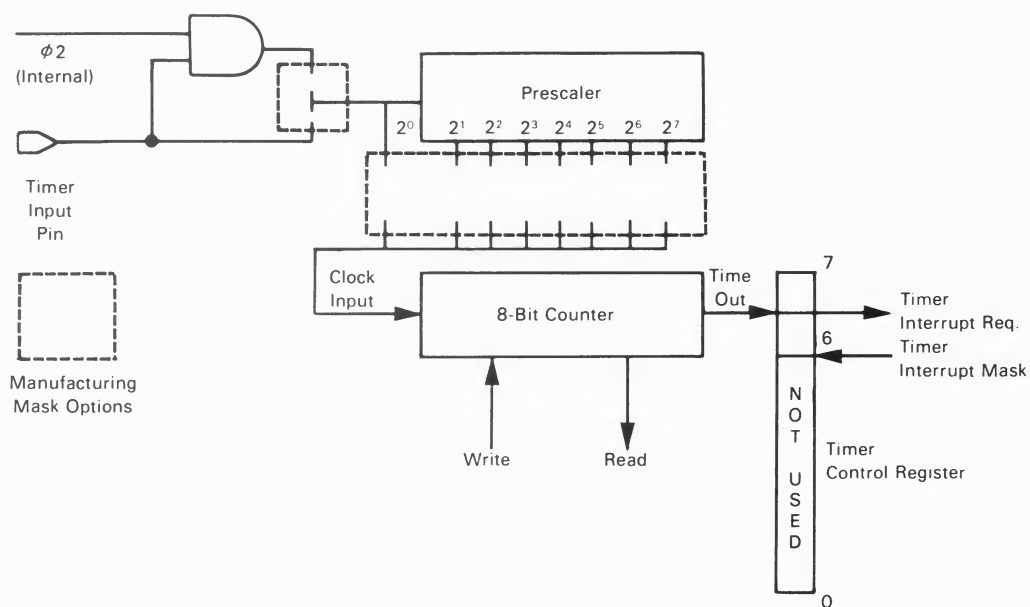
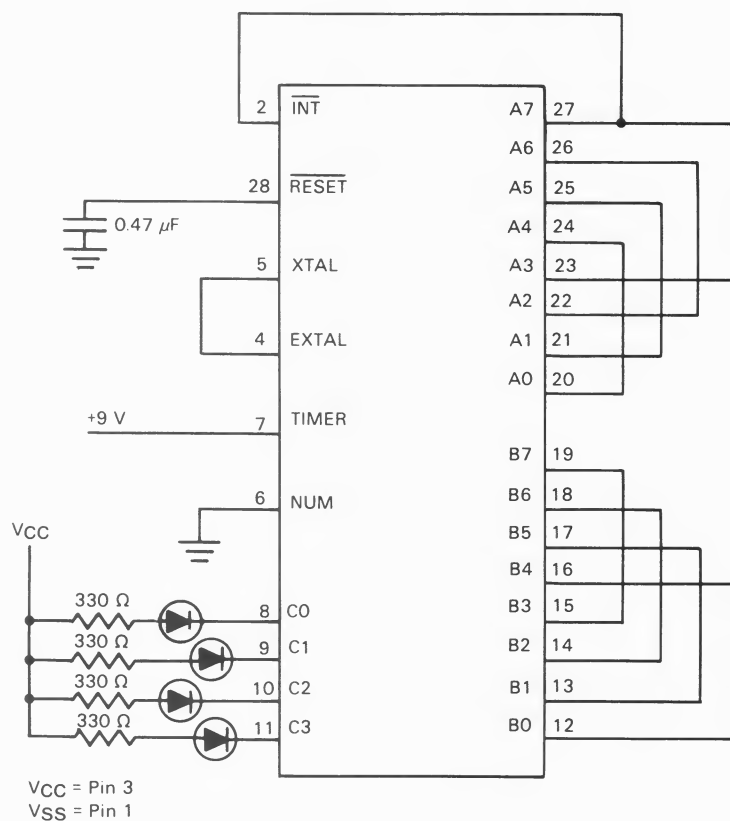


FIGURE 10 — SELF CHECK CONNECTIONS



of the options that has to be specified before manufacture of the MCU. A prescaler option can be applied to the clock input that extends the timing interval up to a maximum of 128 counts before being applied to the counter. This prescaling option must also be specified before manufacturing begins. The timer continues to count past zero and its present count can be monitored at any time by monitoring the timer data register. This allows a program to determine the length of time since a timer interrupt has occurred and not disturb the counting process.

At power up or reset the prescaler and counter are initialized with all logical ones; the timer interrupt request bit (bit 7) is cleared and the timer interrupt request mask bit (bit 6) is set.

SELF CHECK

The self check capability of the MCU provides an internal check to determine if the part is functional. Connect the MCU as shown in Figure 10 and monitor the output of port C bit 3 for an oscillation of approximately three hertz.

RESETS

The MCU can be reset three ways: by the external reset input ($\overline{\text{RESET}}$), by an internal low voltage detect circuit, and during the power up time. See Figure 11.

Upon power up, a minimum of 20 milliseconds is needed before allowing the reset input to go high. This time allows the internal oscillator to stabilize. Connecting a capacitor to the $\overline{\text{RESET}}$ input as shown in Figure 12 will provide sufficient delay.

INTERNAL OSCILLATOR OPTIONS

The internal oscillator circuit has been designed to require a minimum of external components. The use of a crystal (AT cut, 4 MHz max) or a resistor is sufficient to drive the internal oscillator with varying degrees of stability. A manufacturing mask option is available to provide better matching between the external components and the internal oscillator.

The different connection methods are shown in Figure 13. Crystal specifications are given in Figure 14. A resistor selection graph is given in Figure 15.

FIGURE 11 — POWER UP AND RESET TIMING

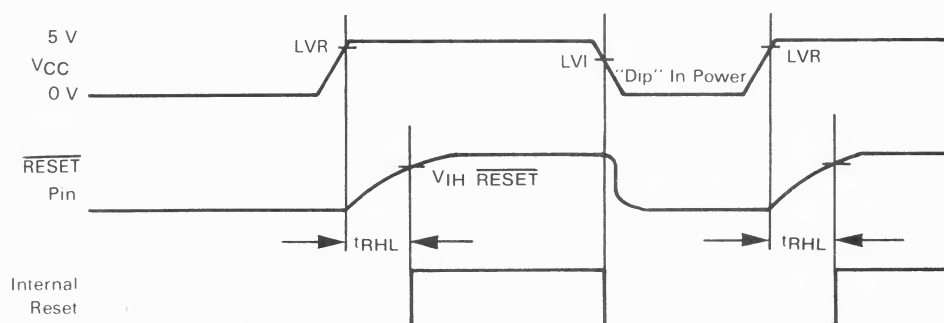


FIGURE 12 — POWER UP RESET DELAY CIRCUIT

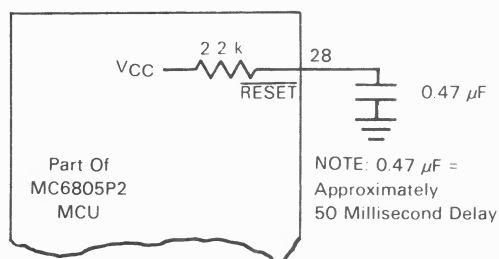
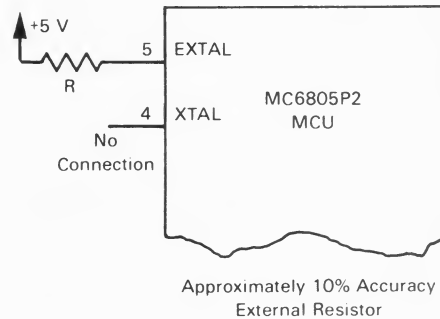
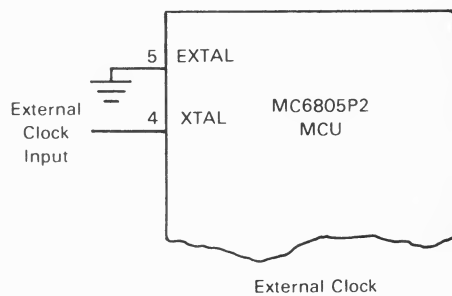
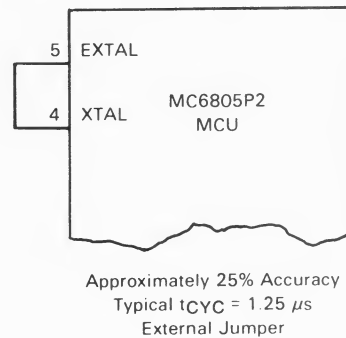
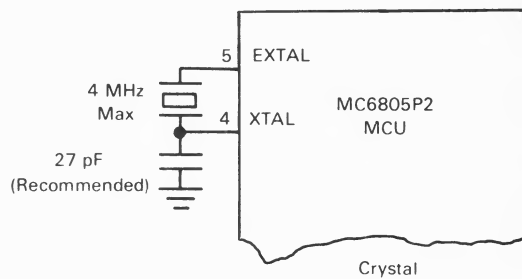


FIGURE 13 — INTERNAL OSCILLATOR OPTIONS

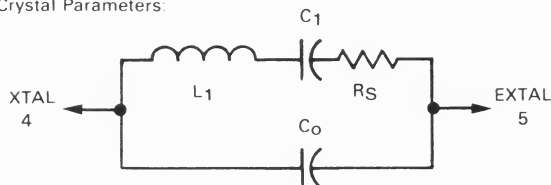


CRYSTAL OPTIONS

RESISTOR OPTIONS

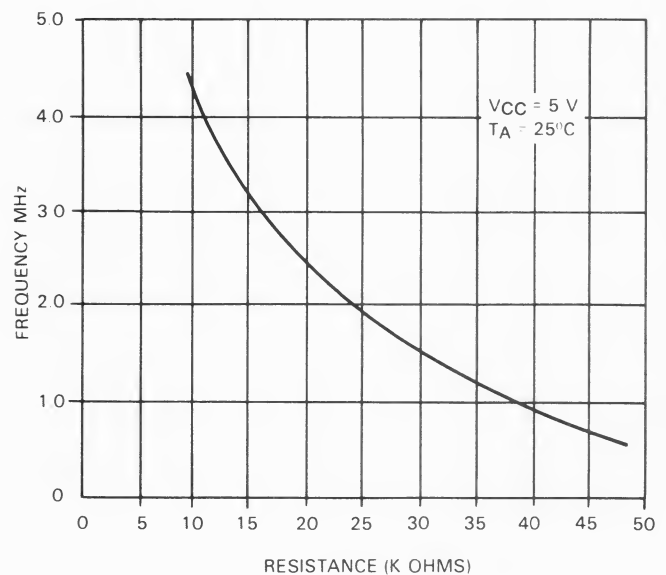
FIGURE 14 — CRYSTAL PARAMETERS

Crystal Parameters:



AT — Cut Parallel Resonance Crystal
 $C_0 = 7 \text{ pF Max}$
 $\text{FREQ} = 4.0 \text{ MHz @ } C_L = 24 \text{ pF}$
 $R_S = 50 \text{ ohms Max.}$

FIGURE 15 — TYPICAL RESISTOR SELECTION GRAPH



INTERRUPTS

The MCU can be interrupted three different ways: through the external interrupt ($\overline{\text{INT}}$) input pin, the internal timer interrupt request, and a software interrupt instruction (SWI). When any interrupt occurs, processing is suspended, the present MCU state is pushed onto the stack, the interrupt bit (I) in the condition code register is set, the address of the interrupt routine is obtained from the appropriate interrupt vector address, and the interrupt routine is executed. The interrupt service routines normally end with a return from interrupt (RTI) instruction which allows the MCU to resume processing of the program prior to the interrupt. Table 1 provides a listing of the interrupts, their priority, and the vector address that contain the starting address of the appropriate interrupt routine.

A sinusoidal signal (1 kHz maximum) can be used to generate an external interrupt ($\overline{\text{INT}}$) as shown in Figure 16.

A flowchart of the interrupt processing sequence is given in Figure 17.

TABLE 1 — INTERRUPT PRIORITIES

Interrupt	Priority	Vector Address
$\overline{\text{RESET}}$	1	\$7FE and \$7FF
SWI	2	\$7FC and \$7FD
$\overline{\text{INT}}$	3	\$7FA and \$7FB
Timer	4	\$7F8 and \$7F9

INPUT/OUTPUT

There are 20 input/output pins. All pins are programmable as either inputs or outputs under software control of the data direction registers. When programmed as outputs, all I/O pins read latched output data regardless of the logic level at the output pin due to output loading (see Figure 18). When port B is programmed for outputs, it is capable of sinking 10 milliamperes on each pin (one volt maximum). All input/output lines are TTL compatible as both inputs and outputs. Port A lines are CMOS compatible as outputs while port B and C lines are CMOS compatible as inputs. Figure 19 provides some examples of port connections.

FIGURE 16 — TYPICAL SINUSODIAL INTERRUPT CIRCUITS

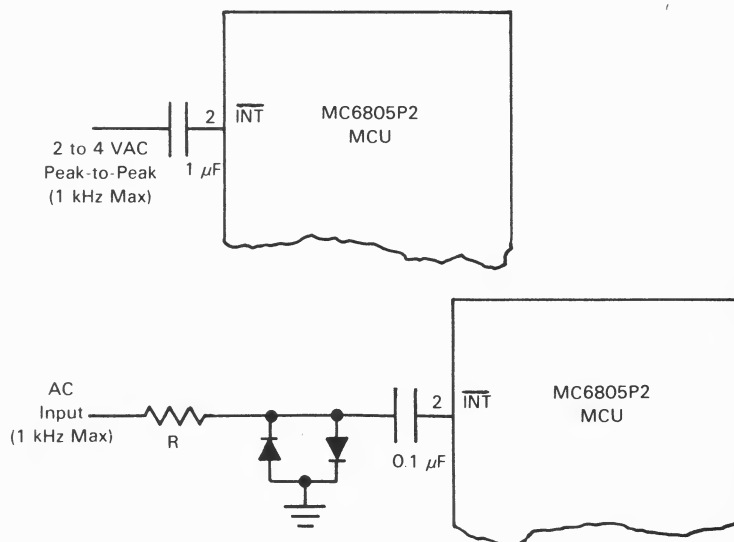


FIGURE 17 — INTERRUPT PROCESSING FLOWCHART

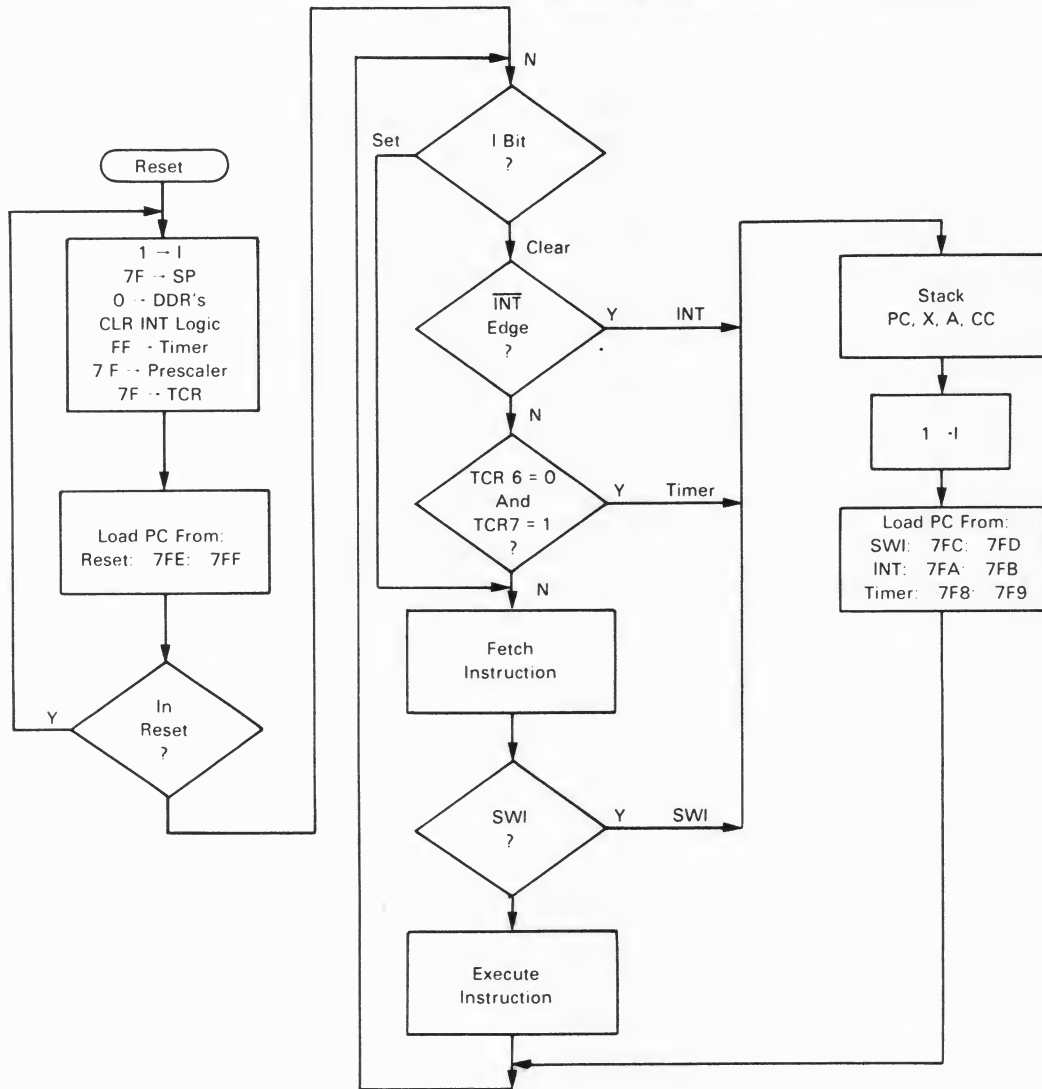
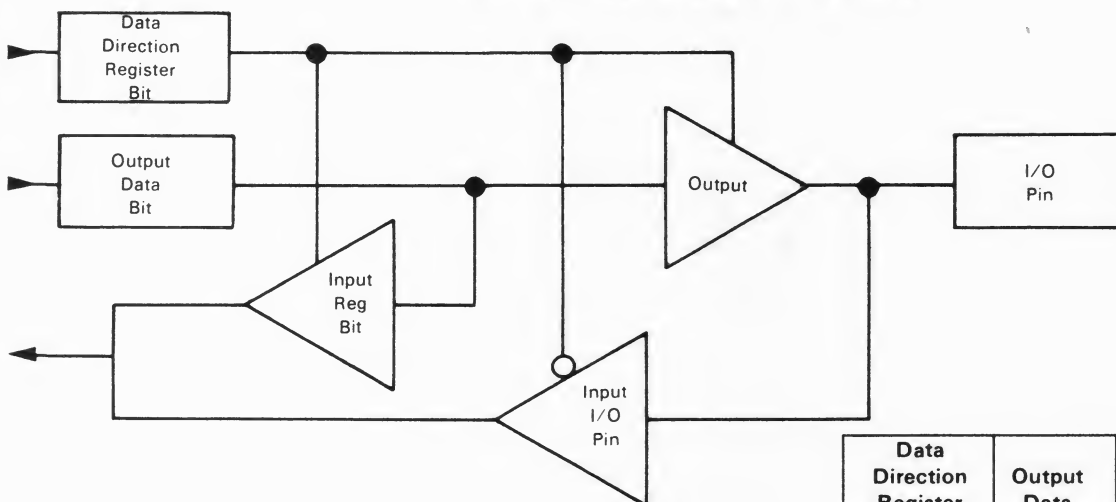


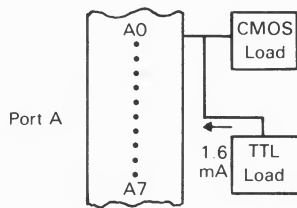
FIGURE 18 — TYPICAL PORT I/O CIRCUITRY



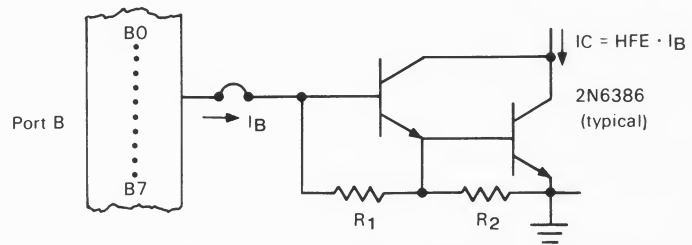
Data Direction Register Bit	Output Data Bit	Output State	Input To MCU
1	0	0	0
1	1	1	1
0	X	3-State	Pin



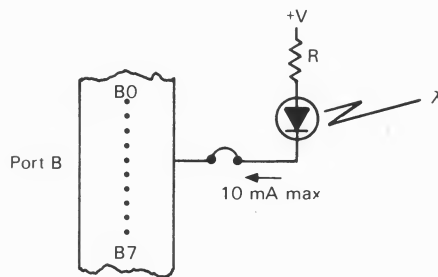
FIGURE 19 — TYPICAL PORT CONNECTIONS



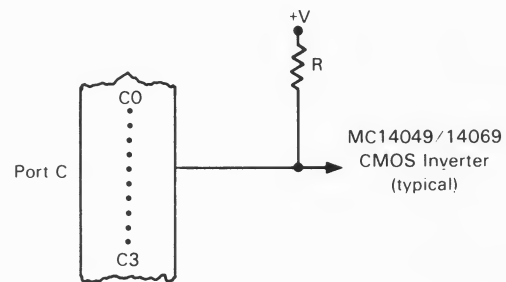
Port A Programmed as output(s) driving CMOS and TTL Load directly
(a)



Port B Programmed as output(s) driving Darlington base directly.
(b)



Port B Programmed as output(s) driving LED(s) directly.
(c)



Port C Programmed as output(s) driving CMOS using external pull-up resistors
(d)

BIT MANIPULATION

The MCU has the ability to set or clear any single random access memory or input/output bit (except the data direction registers) with a single instruction (BSET, BCLR). Any bit in the page zero read only memory can be tested, using the BRSET and BRCLR instructions, and the program branches as a result of its state. This capability to work with any bit in RAM, ROM or I/O allows the user to have individual flags in RAM or to handle single I/O bits as control lines. The example in Figure 20 illustrates the usefulness of the bit manipulation and test instructions. Assume that bit 0 of port A is connected to a zero crossing detector circuit and that bit 1 of port A is connected to the trigger of a TRIAC which powers the controlled hardware.

This program, which uses only seven ROM locations, provides turn-on of the TRIAC within 14 microseconds of the zero crossing. The timer could also be incorporated to

provide turn-on at some later time which would permit pulse-width modulation of the controlled power.

ADDRESSING MODES The MCU has ten addressing modes available for use by the programmer. They are explained and illustrated briefly in the following paragraphs.

FIGURE 20 — BIT MANIPULATION EXAMPLE

```

      .
      .
      .
SELF 1, BRCLR 0, PORTA, SELF 1
      . BSET 1, PORTA
      . BCLR 1, PORTA
      .
      .
      .

```



IMMEDIATE — Refer to Figure 21. The immediate addressing mode accesses constants which do not change during program execution. Such instructions are two bytes long. The effective address (EA) is the PC and the operand is fetched from the byte following the opcode.

DIRECT — Refer to Figure 22. In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in memory. All RAM space, I/O registers and 128 bytes of ROM are located in page zero to take advantage of this efficient memory addressing mode.

EXTENDED — Refer to Figure 23. Extended addressing is used to reference any location in memory space. The EA is the contents of the two bytes following the opcode. Extended addressing instructions are three bytes long.

RELATIVE — Refer to Figure 24. The relative addressing mode applies only to the branch instructions. In this mode the contents of the byte following the opcode is added to the program counter when the branch is taken. $EA = (PC) + 2 + Rel$. Rel is the contents of the location following the instruction opcode with bit 7 being the sign bit. If the branch is not taken $Rel = 0$, when a branch takes place, the program goes to somewhere within the range of +129 bytes to -127 of the present instruction. These instructions are two bytes long.

INDEXED (NO OFFSET) — Refer to Figure 25. This mode of addressing accesses the lowest 256 bytes of memory. These instructions are one byte long and their EA is the contents of the index register.

INDEXED (8-BIT OFFSET) — Refer to Figure 26. The EA is calculated by adding the contents of the byte

following the opcode to the contents of the index register. In this mode, 511 low memory locations are accessible. These instructions occupy two bytes.

INDEXED (16-BIT OFFSET) — Refer to Figure 27. This addressing mode calculates the EA by adding the contents of the two bytes following the opcode to the index register. Thus, the entire memory space may be accessed. Instructions which use this addressing mode are three bytes long.

BIT SET/CLEAR — Refer to figure 28. This mode of addressing applies to instructions which can set or clear any bit on page zero. The lower three bits in the opcode specify the bit to be set or cleared while the byte following the opcode specifies the address in page zero.

BIT TEST AND BRANCH — Refer to Figure 29. This mode of addressing applies to instructions which can test any bit in the first 256 locations (\$00-\$FF) and branch to any location relative to the PC. The byte to be tested is addressed by the byte following the opcode. The individual bit within that byte to be tested is addressed by the lower three bits of the opcode. The third byte is the relative address to be added to the program counter if the branch condition is met. These instructions are three bytes long. The value of the bit tested is written to the carry bit in the condition code register.

INHERENT — Refer to Figure 30. The inherent mode of addressing has no EA. All the information necessary to execute an instruction is contained in the opcode. Direct operations on the accumulator and the index register are included in this mode of addressing. In addition, control instructions such as SWI RTI belong to this group. All inherent addressing instructions are one byte long.

FIGURE 21 — IMMEDIATE ADDRESSING EXAMPLE

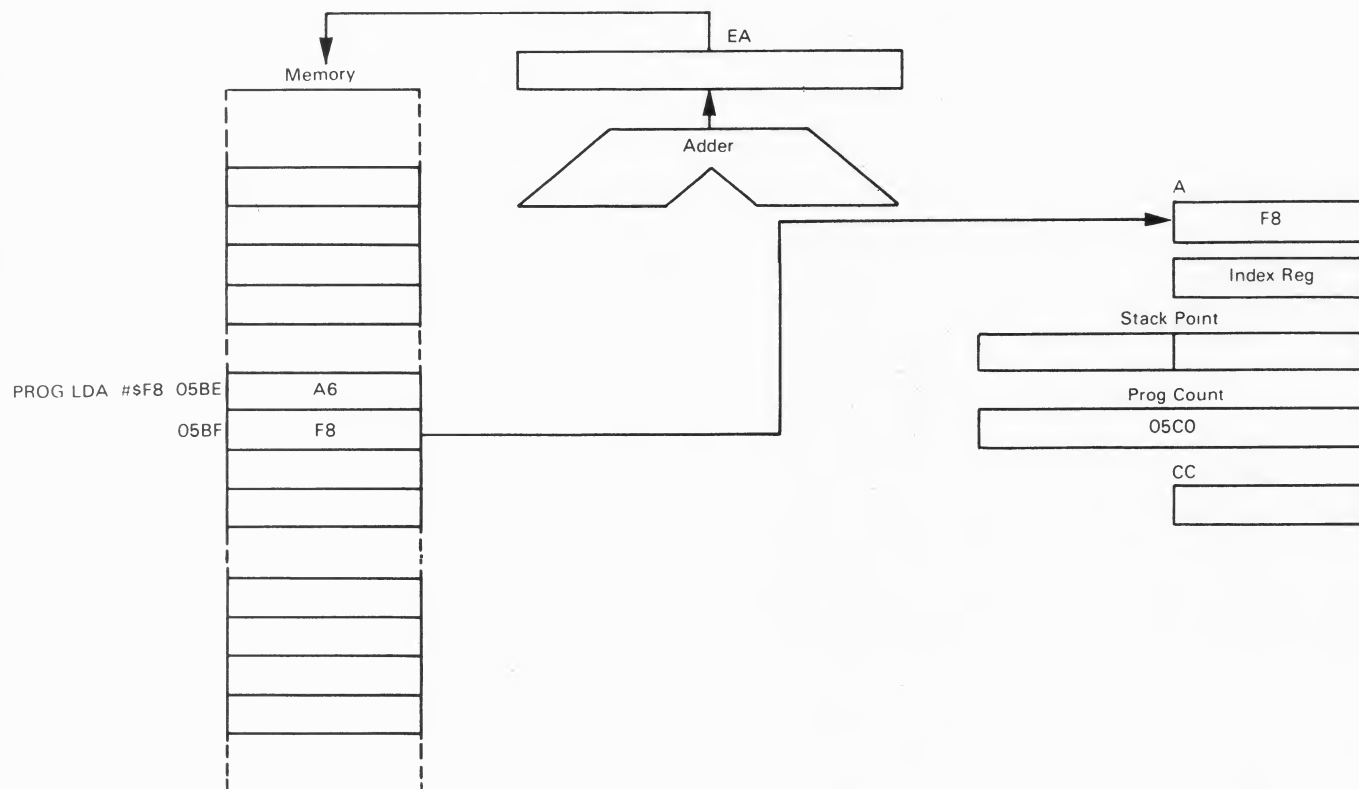


FIGURE 22 — DIRECT ADDRESSING EXAMPLE

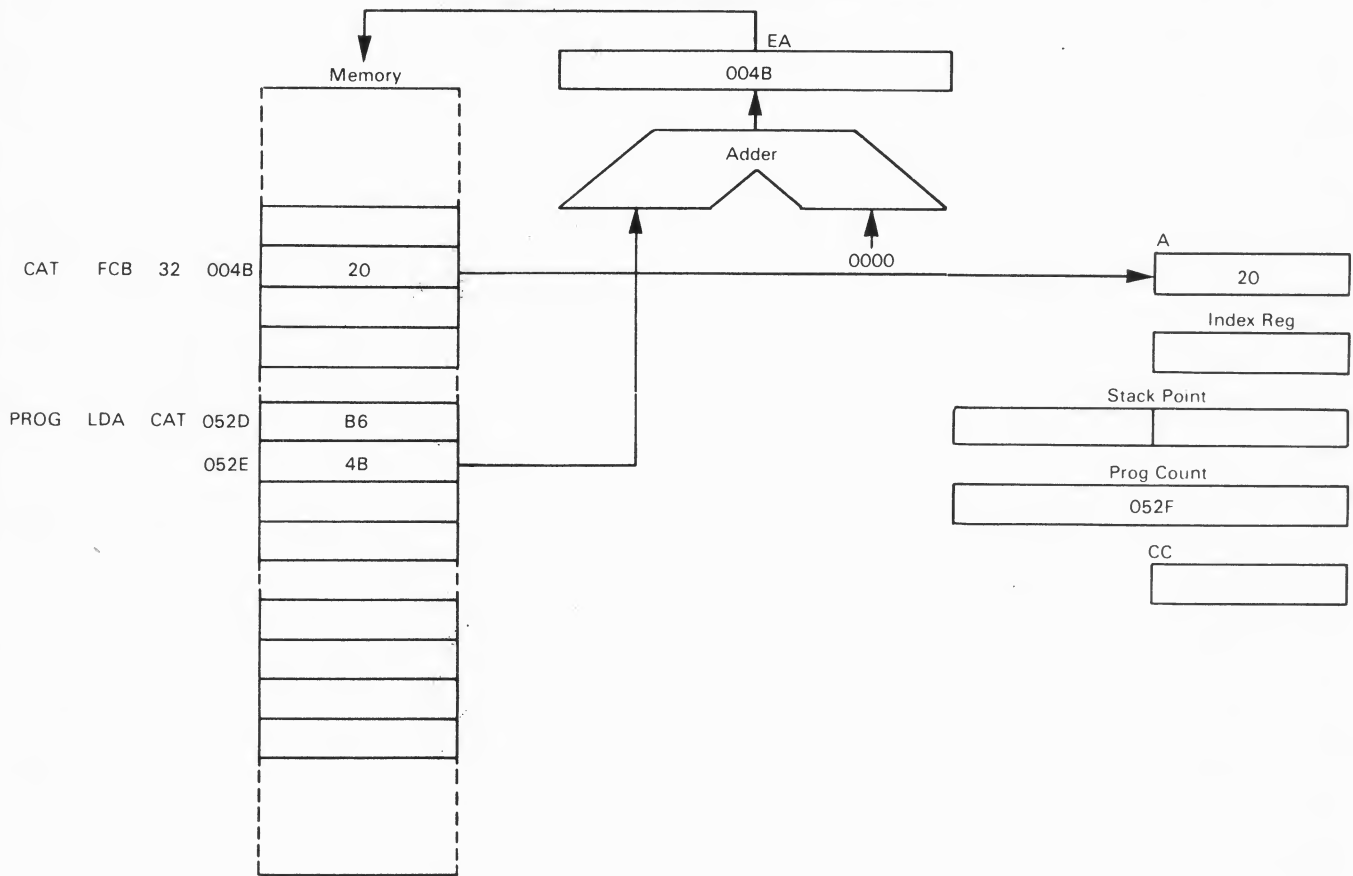


FIGURE 23 — EXTENDED ADDRESSING EXAMPLE

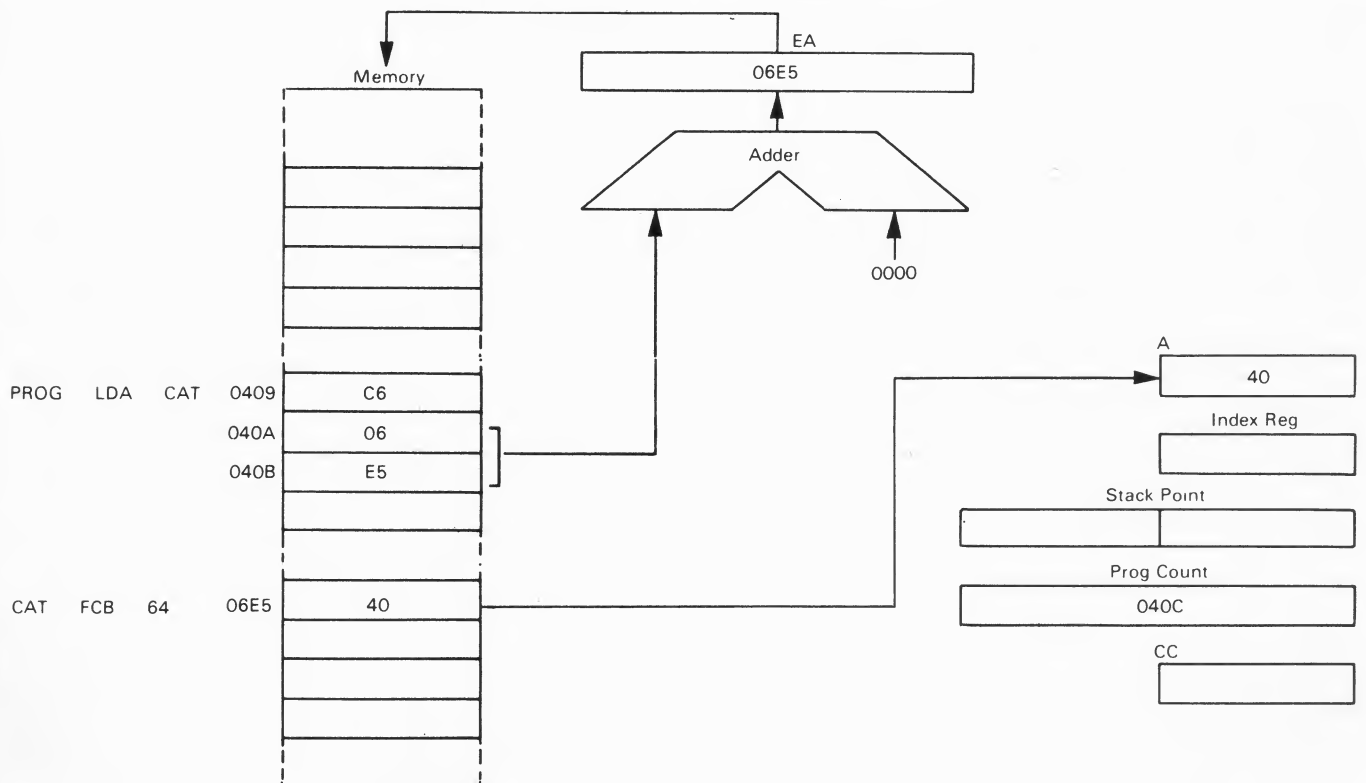


FIGURE 24 — RELATIVE ADDRESSING EXAMPLE

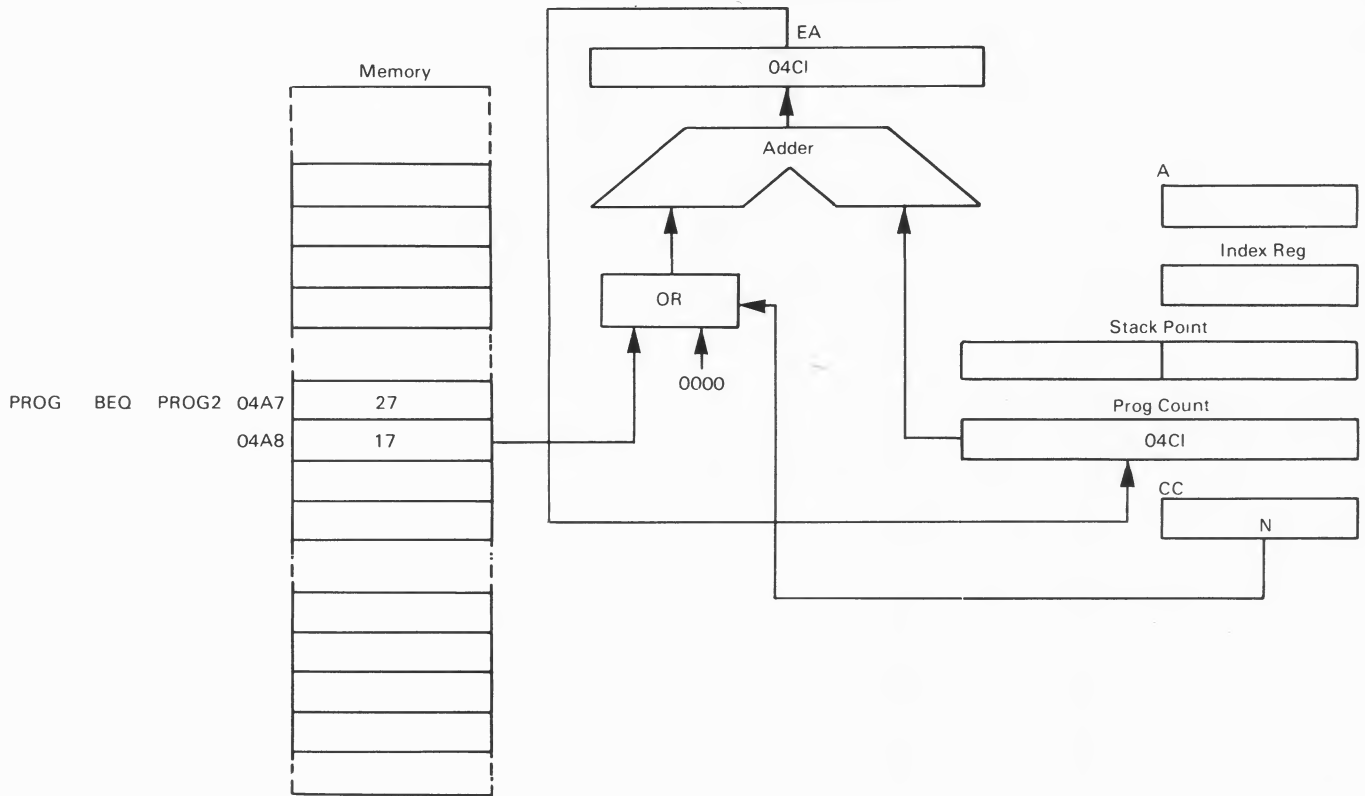


FIGURE 25 — INDEXED (NO OFFSET) ADDRESSING EXAMPLE

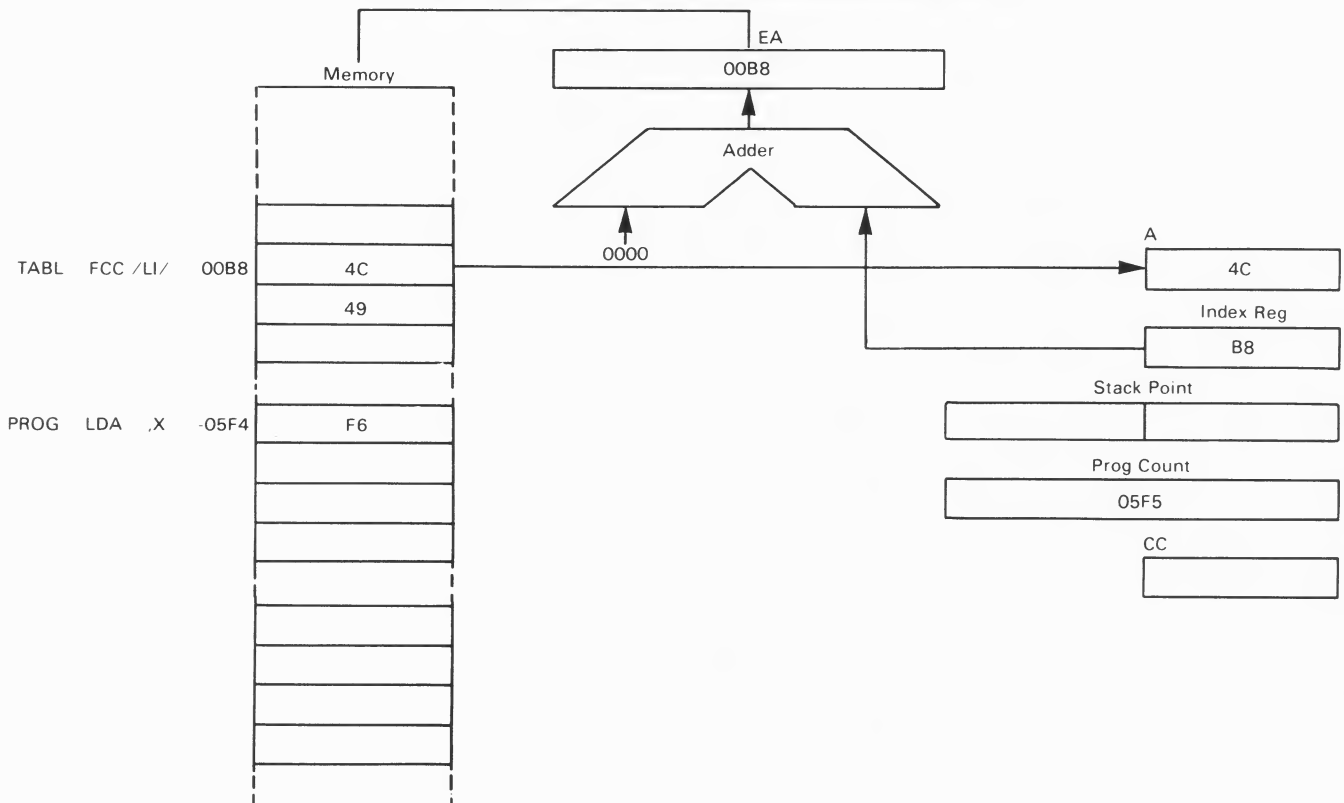


FIGURE 26 — INDEXED (8-BIT OFFSET) ADDRESSING EXAMPLE

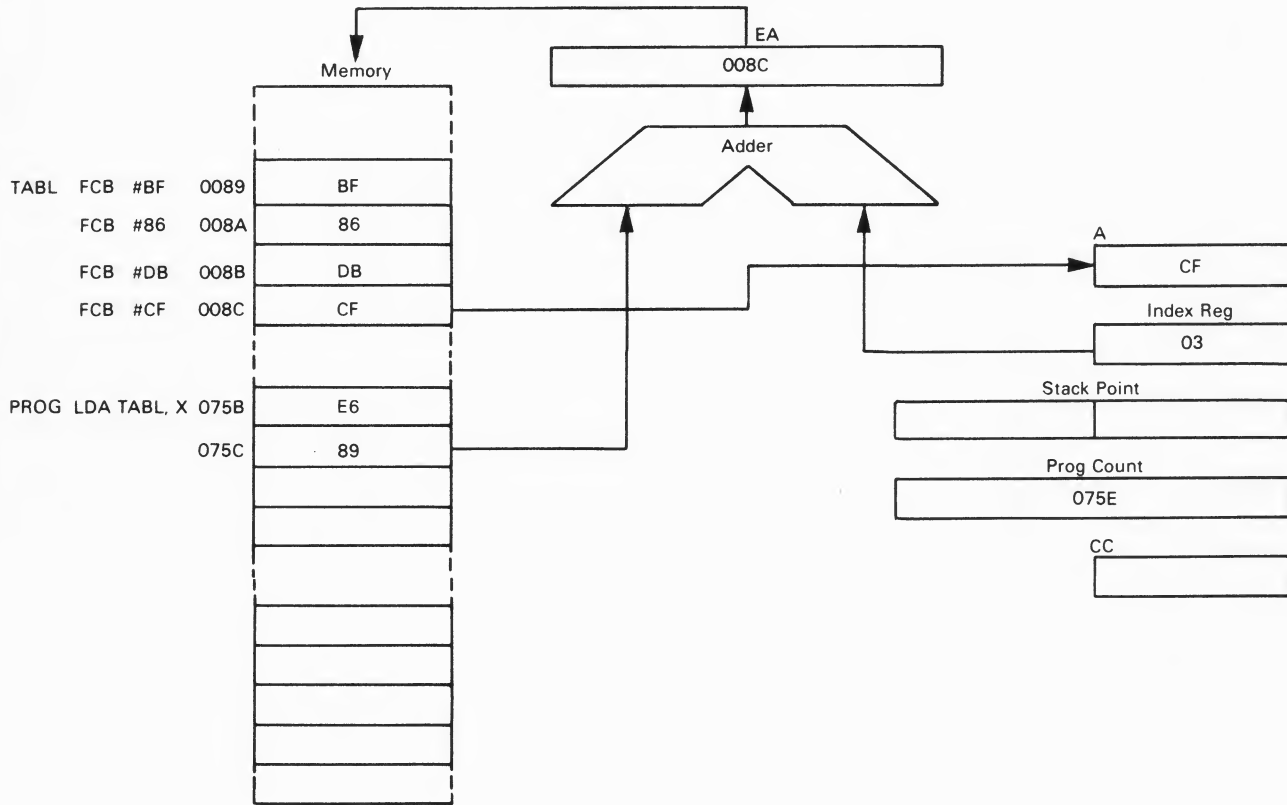


FIGURE 27 — INDEXED (16-BIT OFFSET) ADDRESSING EXAMPLE

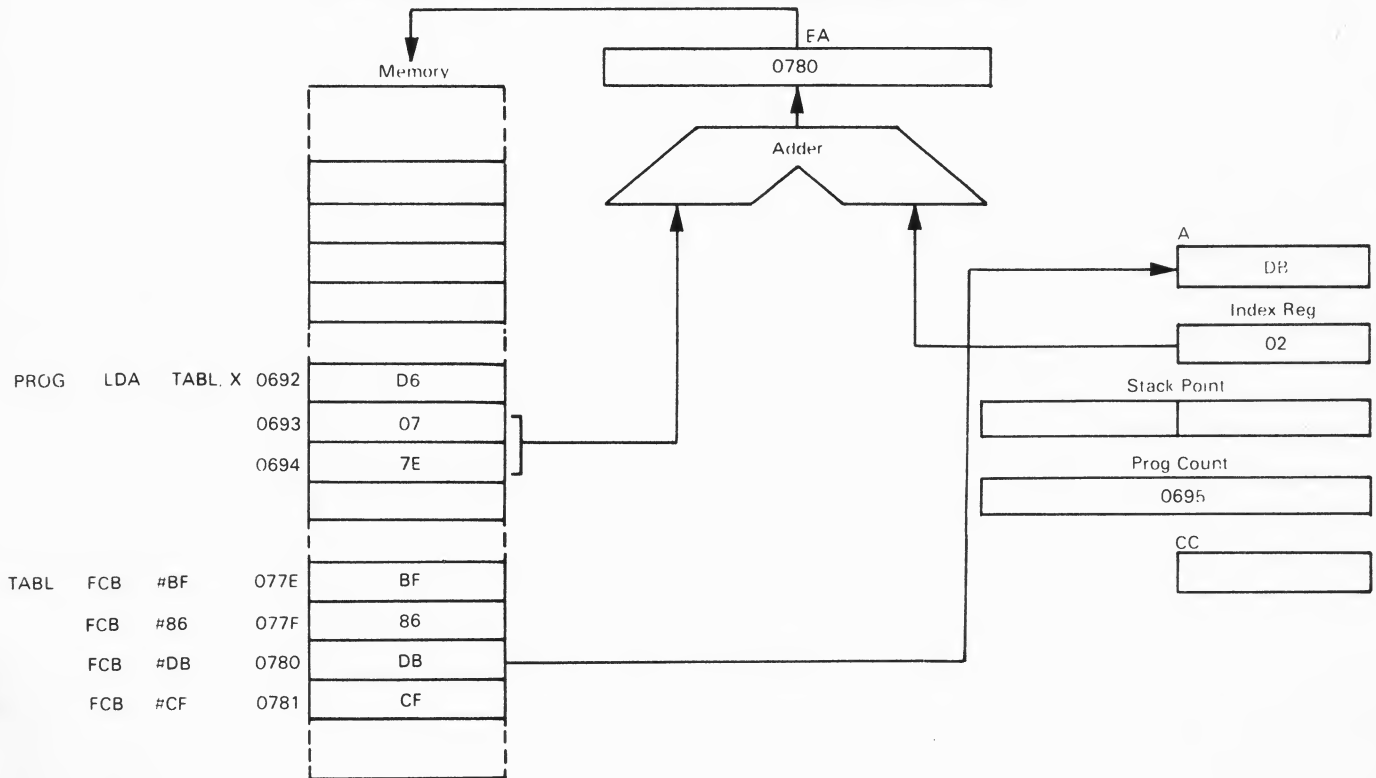


FIGURE 28 — BIT SET/CLEAR ADDRESSING EXAMPLE

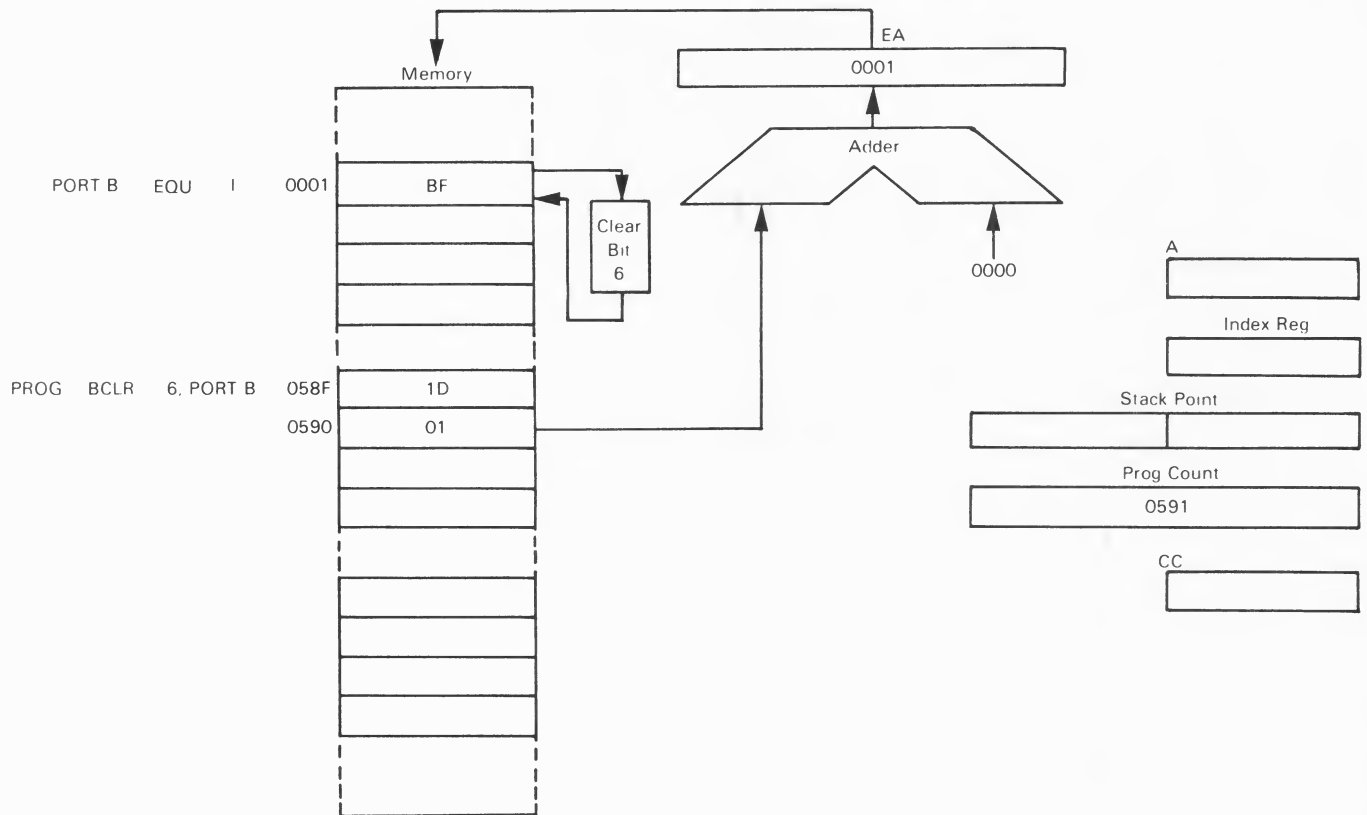


FIGURE 29 — BIT TEST AND BRANCH ADDRESSING EXAMPLE

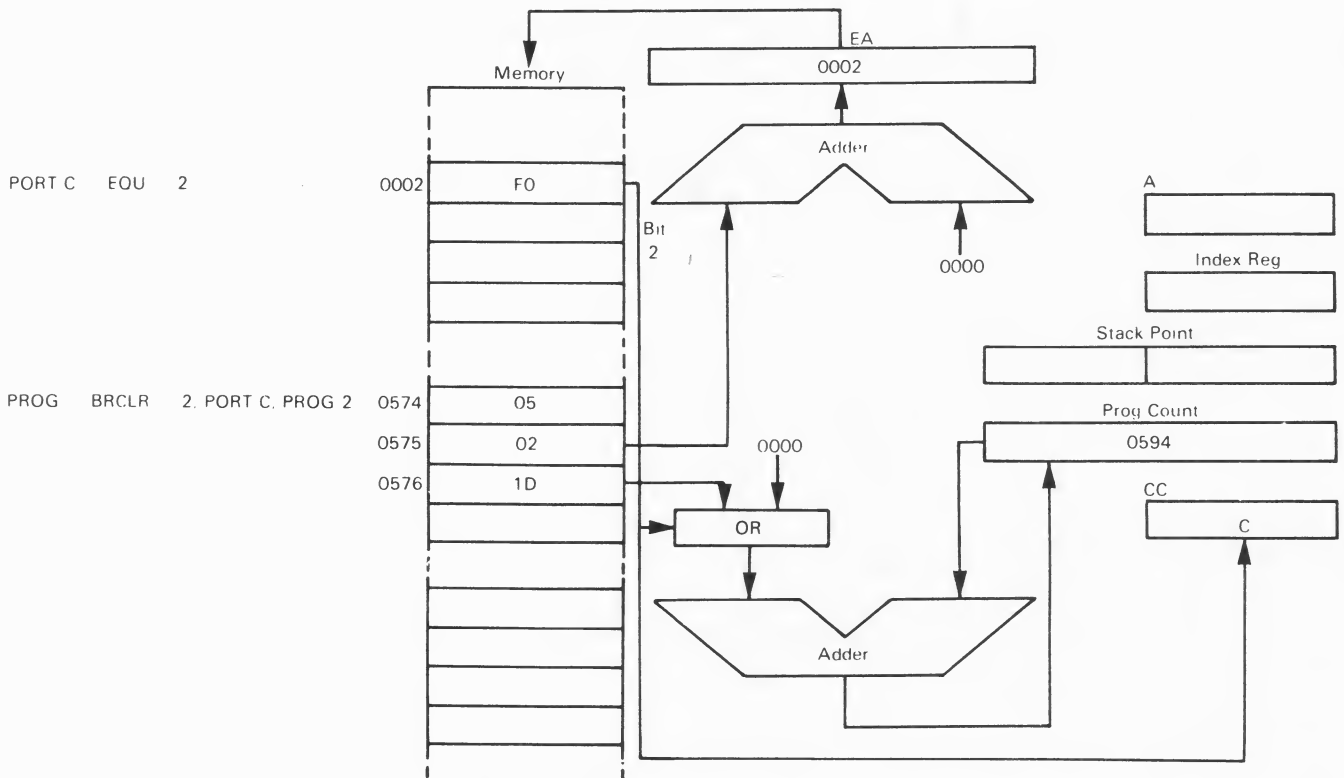
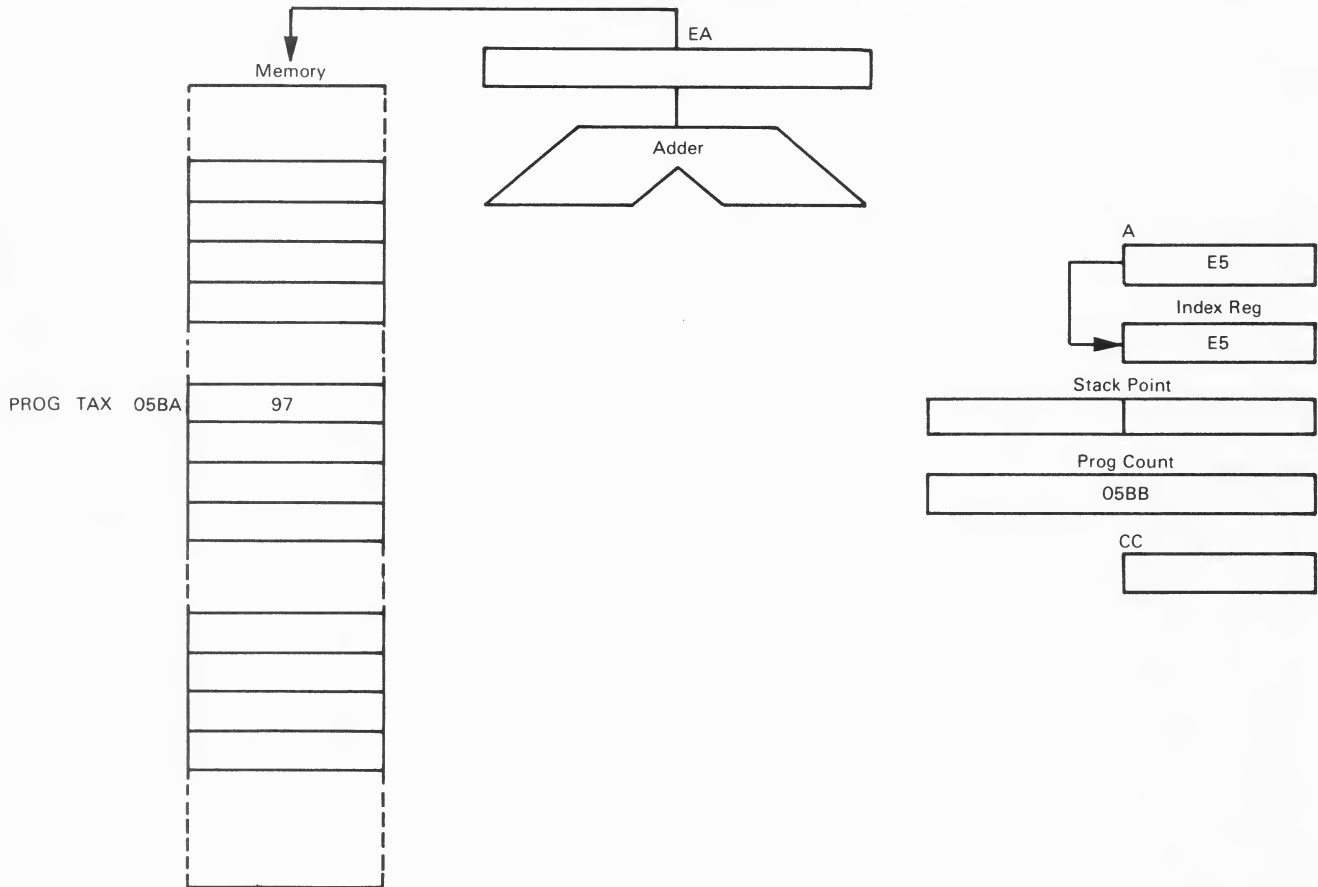


FIGURE 30 — INHERENT ADDRESSING EXAMPLE



INSTRUCTION SET

The MCU has a set of 59 basic instructions. They can be divided into five different types: register/memory, read/modify/write, branch, bit manipulation, and control. The following paragraphs briefly explain each type. All the instructions within a given type are presented in individual tables.

REGISTER/MEMORY INSTRUCTIONS — Most of these instructions use two operands. One operand is either the accumulator or the index register. The other operand is obtained from memory using one of the addressing modes. The jump unconditional (JMP) and jump to subroutine (JSR) instructions have no register operand. Refer to Table 2.

READ/MODIFY/WRITE INSTRUCTIONS — These instructions read a memory location or a register, modify or test its contents, and write the modified value back to memory or to the register. The test for negative or zero (TST) instruction is an exception to the read/modify/

write instructions since it does not perform the write. Refer to Table 3.

BRANCH INSTRUCTIONS — The branch instructions cause a branch from the program when a certain condition is met. Refer to Table 4.

BIT MANIPULATION INSTRUCTIONS — These instructions are used on any bit in the first 256 bytes of the memory. One group either sets or clears. The other group performs the bit test and branch operations. Refer to Table 5.

CONTROL INSTRUCTIONS — The control instructions control the MCU operations during program execution. Refer to Table 6.

ALPHABETICAL LISTING — The complete instruction set is given in alphabetical order in Table 7.

OPCODE MAP — Table 8 is an opcode map for the instructions used on the MCU.



TABLE 2 — REGISTER/MEMORY INSTRUCTIONS

Addressing Modes																				
Function		Immediate			Direct			Extended			Indexed (No Offset)			Indexed (8-Bit Offset)			Indexed (16-Bit Offset)			
		Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles	
Load A from Memory	Mnemonic	LDA	A6	2	2	B6	2	4	C6	3	5	F6	1	4	E6	2	5	D6	3	6
		LDX	AE	2	2	BE	2	4	CE	3	5	FE	1	4	EE	2	5	DE	3	6
Store A in Memory		STA	—	—	—	B7	2	5	C7	3	6	F7	1	5	E7	2	6	D7	3	7
Store X in Memory		STX	—	—	—	BF	2	5	CF	3	6	FF	1	5	EF	2	6	DF	3	7
Add Memory to A		ADD	AB	2	2	BB	2	4	CB	3	5	FB	1	4	EB	2	5	DB	3	6
Add Memory and Carry to A		ADC	A9	2	2	B9	2	4	C9	3	5	F9	1	4	E9	2	5	D9	3	6
Subtract Memory		SUB	A0	2	2	B0	2	4	C0	3	5	F0	1	4	E0	2	5	D0	3	6
Subtract Memory from A with Borrow		SBC	A2	2	2	B2	2	4	C2	3	5	F2	1	4	E2	2	5	D2	3	6
AND Memory to A		AND	A4	2	2	B4	2	4	C4	3	5	F4	1	4	E4	2	5	D4	3	6
OR Memory with A		ORA	AA	2	2	BA	2	4	CA	3	5	FA	1	4	EA	2	5	DA	3	6
Exclusive OR Memory with A		EOR	A8	2	2	B8	2	4	C8	3	5	F8	1	4	E8	2	5	D8	3	6
Arithmetic Compare A with Memory		CMP	A1	2	2	B1	2	4	C1	3	5	F1	1	4	F1	2	5	D1	3	6
Arithmetic Compare X with Memory		CPX	A3	2	2	B3	2	4	C3	3	5	F3	1	4	E3	2	5	D3	3	6
Bit Test Memory with A (Logical Compare)		BIT	A5	2	2	B5	2	4	C5	3	5	F5	1	4	E5	2	5	D5	3	6
Jump Unconditional		JMP	—	—	—	BC	2	3	CC	3	4	FC	1	3	EC	2	4	DC	3	5
Jump to Subroutine		JSR	—	—	—	BD	2	7	CD	3	8	FD	1	7	ED	2	8	DD	3	9

TABLE 3 — READ-MODIFY-WRITE INSTRUCTIONS

Addressing Modes																	
Function		Inherent (A)			Inherent (X)			Direct			Indexed (No Offset)			Indexed (8 Bit Offset)			
		Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles	# Cycles	Op Code	# Bytes	# Cycles	# Cycles	Op Code	# Bytes	# Cycles		
Complement	Increment	INC	4C	1	4	5C	1	4	3C	2	6	7C	1	6	6C	2	7
	Decrement	DEC	4A	1	4	5A	1	4	3A	2	6	7A	1	6	6A	2	7
	Clear	CLR	4F	1	4	5F	1	4	3F	2	6	7F	1	6	6F	2	7
Negate (2's Complement)	Complement	COM	43	1	4	53	1	4	33	2	6	73	1	6	63	2	7
	Negate	NEG	40	1	4	50	1	4	30	2	6	70	1	6	60	2	7
	Rotate Left Thru Carry	ROL	49	1	4	59	1	4	39	2	6	79	1	6	69	2	7
Logical Shift Left	Rotate Right Thru Carry	ROR	46	1	4	56	1	4	36	2	6	76	1	6	66	2	7
	Logical Shift Left	LSL	48	1	4	58	1	4	38	2	6	78	1	6	68	2	7
	Logical Shift Right	LSR	44	1	4	54	1	4	34	2	6	74	1	6	64	2	7
Test for Negative or Zero	Arithmetic Shift Right	ASR	47	1	4	57	1	4	37	2	6	77	1	6	67	2	7
	TST	4D	1	4	5D	1	4	3D	2	6	7D	1	6	6D	2	7	



TABLE 4 — BRANCH INSTRUCTIONS

Function	Mnemonic	Relative Addressing Mode		
		Op Code	# Bytes	# Cycles
Branch Always	BRA	20	2	4
Branch Never	BRN	21	2	4
Branch IFF Higher	BHI	22	2	4
Branch IFF Lower or Same	BLS	23	2	4
Branch IFF Carry Clear	BCC	24	2	4
(Branch IFF Higher or Same)	(BHS)	24	2	4
Branch IFF Carry Set	BCS	25	2	4
(Branch IFF Lower)	(BLO)	25	2	4
Branch IFF Not Equal	BNE	26	2	4
Branch IFF Equal	BEQ	27	2	4
Branch IFF Half Carry Clear	BHCC	28	2	4
Branch IFF Half Carry Set	BHCS	29	2	4
Branch IFF Plus	BPL	2A	2	4
Branch IFF Minus	BMI	2B	2	4
Branch IFF Interrupt Mask Bit is Clear	BMC	2C	2	4
Branch IFF Interrupt Mask Bit is Set	BMS	2D	2	4
Branch IFF Interrupt Line is Low	BIL	2E	2	4
Branch IFF Interrupt Line is High	BIH	2F	2	4
Branch to Subroutine	BSR	AD	2	8

TABLE 5 — BIT MANIPULATION INSTRUCTIONS

Function	Mnemonic	Addressing Modes					
		Bit Set / Clear			Bit Test and Branch		
		Op Code	# Bytes	# Cycles	Op Code	# Bytes	# Cycles
Branch IFF Bit n is set	BRSET n (n = 0...7)	—	—	—	2 · n	3	10
Branch IFF Bit n is clear	BRCLR n (n = 0...7)	—	—	—	01 + 2 · n	3	10
Set Bit n	BSET n (n = 0...7)	10 + 2 · n	2	7	—	—	—
Clear bit n	BCLR n (n = 0...7)	11 + 2 · n	2	7	—	—	—

TABLE 6 — CONTROL INSTRUCTIONS

Function	Mnemonic	Inherent		
		Op Code	# Bytes	# Cycles
Transfer A to X	TAX	97	1	2
Transfer X to A	TXA	9F	1	2
Set Carry Bit	SEC	99	1	2
Clear Carry Bit	CLC	98	1	2
Set Interrupt Mask Bit	SEI	9B	1	2
Clear Interrupt Mask Bit	CLI	9A	1	2
Software Interrupt	SWI	83	1	11
Return from Subroutine	RTS	81	1	6
Return from Interrupt	RTI	80	1	9
Reset Stack Pointer	RSP	9C	1	2
No-Operation	NOP	9D	1	2



TABLE 7 — INSTRUCTION SET

Mnemonic	Addressing Modes										Condition Code				
	Inherent	Immediate	Direct	Extended	Relative	Indexed (No Offset)	Indexed (8 Bits)	Indexed (16 Bits)	Bit Set/ Clear	Bit Test & Branch	H	I	N	Z	C
ADC		X	X	X		X	X	X			Λ	●	Λ	Λ	Λ
ADD		X	X	X		X	X	X			Λ	●	Λ	Λ	Λ
AND		X	X	X		X	X	X			●	●	Λ	Λ	●
ASL	X		X			X	X				●	●	Λ	Λ	Λ
ASR	X		X			X	X				●	●	Λ	Λ	Λ
BCC					X						●	●	●	●	●
BCLR									X		●	●	●	●	●
BCS					X						●	●	●	●	●
BEQ					X						●	●	●	●	●
BHCC					X						●	●	●	●	●
BHCS					X						●	●	●	●	●
BHI					X						●	●	●	●	●
BHS					X						●	●	●	●	●
BIH					X						●	●	●	●	●
BIL					X						●	●	●	●	●
BIT		X	X	X		X	X	X			●	●	Λ	Λ	●
BLO					X						●	●	●	●	●
BLS					X						●	●	●	●	●
BMC					X						●	●	●	●	●
BMI											●	●	●	●	●
BMS											●	●	●	●	●
BNE											●	●	●	●	●
BPL											●	●	●	●	●
BRA											●	●	●	●	●
BRN											●	●	●	●	●
BRCLR										X	●	●	●	●	Λ
BRSET										X	●	●	●	●	Λ
BSET									X		●	●	●	●	●
BSR					X						●	●	●	●	●
CLC	X										●	●	●	●	0
CLI	X										●	0	●	●	●
CLR	X		X			X	X				●	●	0	1	●
CMP		X	X	X		X	X	X			●	●	Λ	Λ	Λ
COM	X		X			X	X				●	●	Λ	Λ	1
CPX		X	X	X		X	X	X			●	●	Λ	Λ	Λ
DEC	X		X			X	X				●	●	Λ	Λ	●
EOR		X	X	X		X	X	X			●	●	Λ	Λ	●
INC	X		X			X	X				●	●	Λ	Λ	●
JMP			X	X		X	X	X			●	●	●	●	●
JSR			X	X		X	X	X			●	●	●	●	●
LDA		X	X	X		X	X	X			●	●	Λ	Λ	●
LDX		X	X	X		X	X	X			●	●	Λ	Λ	●
LSL	X		X			X	X				●	●	Λ	Λ	Λ
LSR	X		X			X	X				●	●	0	Λ	Λ
NEQ	X		X			X	X				●	●	Λ	Λ	Λ
NOP	X										●	●	●	●	●
ORA		X	X	X		X	X	X			●	●	Λ	Λ	●
ROL	X		X			X	X				●	●	Λ	Λ	Λ
RSP	X										●	●	●	●	●

Condition Code Symbols:

H	Half Carry (From Bit 3)	C	Carry Borrow
I	Interrupt Mask	Λ	Test and Set if True, Cleared Otherwise
N	Negative (Sign Bit)	●	Not Affected
Z	Zero		



TABLE 7 — INSTRUCTION SET
(CONT.)

Mnemonic	Addressing Modes										Condition Code				
	Inherent	Immediate	Direct	Extended	Relative	Indexed (No Offset)	Indexed (8 Bits)	Indexed (16 Bits)	Bit Set/ Clear	Bit Test & Branch	H	I	N	Z	C
RTI	X										?	?	?	?	?
RTS	X										●	●	●	●	●
SBC		X	X	X		X	X	X			●	●	△	△	△
SEC	X										●	●	●	●	1
SEI	X										●	1	●	●	●
STA			X	X		X	X	X			●	●	△	△	●
STX			X	X		X	X	X			●	●	△	△	●
SUB		X	X	X		X	X	X			●	●	△	△	△
SWI	X										●	1	●	●	●
TAX	X										●	●	●	●	●
TST	X		X			X	X				●	●	△	△	●
TXA	X										●	●	●	●	●

Condition Code Symbols:

- | | |
|---------------------------|-------------------------------------------|
| H Half Carry (From Bit 3) | C Carry/Borrow |
| I Interrupt Mask | △ Test and Set if True, Cleared Otherwise |
| N Negative (Sign Bit) | ● Not Affected |
| Z Zero | ? Load CC Register From Stack |

TABLE 8 — OPCODE MAP

Bit Manipulation		Brnch	Read/Modify/Write						Control		Register/Memory						← High	
Test & Branch	Set/ Clear	Rel	DIR	A	X	,X1	,X0	INH	INH	IMM	DIR	EXT	,X2	,X1	,X0			
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F			
-	BRSET0	BSET0	BRA	NEQ					RTI*	—	SUB						0	LOW
1	BRCLR0	BCLR0	BRN	—					RTS*	—	CMP						1	
2	BRSET1	BSET1	BHI	—					—	—	SBC						2	
3	BRCLR1	BCLR1	BLS	COM					SWI*	—	CMPX/CPX						3	
4	BRSET2	BSET2	BCC	LSR					—	—	AND						4	
5	BRCLR2	BCLR2	BCS	—					—	—	BIT						5	
6	BRSET3	BSET3	BNE	ROR					—	—	LDA						6	
7	BRCLR3	BCLR3	BEQ	ASR					—	TAX	—	STA (+1)					7	
8	BRSET4	BSET4	BHCC	LSL/ASL					—	CLC	EOR						8	
9	BRCLR4	BCLR4	BHCS	ROL					—	SEC	ADC						9	
A	BRSET5	BSET5	BPL	DEC					—	CLI	ORA						A	
B	BRCLR5	BCLR5	BMI	—					—	SEI	ADD						B	
C	BRSET6	BSET6	BMC	INC					—	RSP	—	JMP (-1)					C	
D	BRCLR6	BCLR6	BMS	TST					—	NOP	BSR*	JSR (+3)					D	
E	BRSET7	BSET7	BIL	—					—	—	LDX						E	
F	BRCLR7	BCLR7	BIH	CLR					—	TXA	STX (+1)						F	
	3/10	2/7	2/4	2/6	1/4	1/4	2/7	1/6	1/*	1/2	2/2	2/4	3/5	3/6	2/5	1/4		

Notes:

Undefined opcodes are marked with "—"

The numbers at the bottom of each column denote the number of bytes and the number of cycles required (Bytes/Cycles)

Mnemonics followed by a "*" require a different number of cycles as follows

RTI 9
RTS 6
SWI 11
BSR 8

() indicate that the number in parenthesis must be added to the cycle count for that instruction



ORDERING INFORMATION

The following information is required when ordering a custom MCU. This information may be transmitted to Motorola in any of the following media:

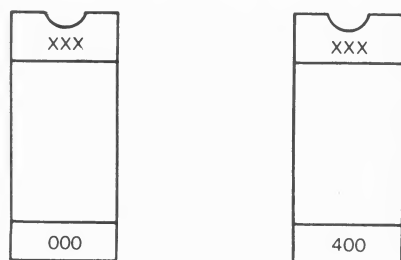
PROM(s)
Assembler formatted object tape
Punched card deck
Paper tape of card deck format
MDOS disk file

To initiate a ROM pattern for the MCU it is necessary to first contact your local field service office, local sales person or your local Motorola representative.

PROMS — The MCM2708 or MCM2716 type PROMs, programmed with the customer program (positive logic sense for address and data), may be submitted for pattern generation. The PROMs must be clearly marked to indicate which PROM corresponds to which address space (000-3FF HEX). See Figure 31 for recommended marking procedure.

After the PROM(s) are marked they should be placed in conductive IC carriers and securely packed. Do not use styrofoam.

FIGURE 31 — PROM MARKING



XXX = Customer ID

ASSEMBLER FORMATTED OBJECT TAPE —

Cassette tapes produced on a Silent 700 terminal and EXORciser are acceptable.

PUNCHED CARD DECK — The custom MCU may be specified for manufacture in the form of standard 80-column punched cards.

The card deck for specifying the custom MCU has the following format:

OPTION CARD
COMMENT CARDS
X CARDS
C CARDS

Option Card — The first card in the deck must be the OPTION CARD. The format is as follows:

Column 1-20: Customer name. Any 20 characters may be used.
Column 25-29: This is a 5-digit number assigned by Motorola. Leave this field blank. It will be punched at Motorola unless otherwise notified.
Column 37-39: Address field base on output listing. The characters HEX or DEC specify the output listing address base as hexadecimal or decimal, respectively. If omitted, DEC is assumed.

Column 41-43: Data field base on output listing. The characters HEX or DEC specify the output listing ROM data base as hexadecimal or decimal, respectively. If omitted, DEC is assumed.

COMMENT CARDS — Comment Cards must have an asterisk (*) in column 1. The remaining 79 columns may contain any letter, number, or character.

X Cards — Five X cards are possible. All X cards have an X in column 1 and one or three or more words, each separated by one blank space.

The possibilities are:

- 1) X SEQUENCE
- 2) X BASE DEC DEC
- 3) X BASE DEC HEX
- 4) X BASE HEX DEC
- 5) X BASE HEX HEX

Card 1 specifies that there are sequence numbers on each data card that follows. The sequence numbers must be in columns 77-79 of the data cards (C Cards) and must be in decimal, right justified. The numbers must start with 1 (one) and must be in order. The X SEQUENCE Card may appear anywhere within the deck after the Option Card. If it appears within the data card section, data cards encountered before the X SEQUENCE Card will not be checked for sequence numbers. All following cards will be checked. If no X SEQUENCE Card is used, no sequence numbers will be checked.

It is initially assumed that the address and byte count as well as the data specified on the C cards will be in decimal. An X BASE card can be used to override this specification. The second word on the card (that following BASE) specifies the base (either DEC or HEX) of the address and byte count on all following C cards. The last word specifies the base of the data fields on the C cards. An X BASE card may appear anywhere within the deck following the OPTION card. It may be overridden by another X BASE card. All data cards (C cards) following an X BASE card will be interpreted as per that X BASE card unless another X BASE card is encountered. If no X BASE cards are used, it is assumed that all fields on the C cards are in decimal.

NOTE:

Once an X SEQUENCE card is encountered, all successive cards will be checked for the proper sequence number; and, unlike X BASE cards, this option cannot thereafter be altered by another X SEQUENCE card.

C Cards — These cards contain the actual ROM data. All fields are right-justified.

Column 1: C (the letter C)
Column 2-9: ADD
Column 10-12: BYTE
Column 14-16: DATA 1
Column 17-19: DATA 2

·
·
·

Columns 76-78: DATA 21
Column 77-79: DATA 22 or SEQUENCE NUMBER

ADD is the address of the first byte of data (DATA 1) contained on that card. Byte is the number of bytes of data to be read from that card. BYTE must be greater than zero



and less than 23 (1-22) if no sequence numbers are used, and less than 22 (1-21) if sequence numbers are used. If, for example, there are ten data fields punched on the card, by BYTE = 2, only the first two will be read. Also, if there are two punched data fields, for example, and BYTE = 6, six ROM locations will be filled from that card. The four unspecified fields will be decoded as zero. ADD and BYTE are always in the same base (HEX or DECIMAL). DATA 1 through DATA N is the data to be placed in the ROM at address ADD through ADD + (N-1), respectively.

Any ROM address not filled as a result of reading data from a C card will be filled with zero. If a particular location has already been specified by a C card, but a successive C card also has the data which is to be placed in that location, the second C card will override the first.

PAPER TAPE OF CARD DECK FORMAT — Punched Paper tape (ASCII) in the same format as cards can also be accepted. However, your order will be processed faster if the data is in card format. After the tape leader there should be a CR LF. Data records should be a full 80 columns, each terminated by a CR LF. Following the last Data record, there should be one more record with the first three characters being EOF, followed by 77 blanks and a CR LF.

CR = Carriage Return

LF = Line Feed

OPTION LIST

Select the options for your MCU from the following list. A manufacturing mask will be generated from this information.

☐ ROM Mask

Timer Clock Source

☐ Internal $\phi 2$ clock

☐ TIMER input pin (7)

Timer Prescaler

☐ 2^0 (divide by 1)

☐ 2^4 (divide by 16)

☐ 2^1 (divide by 2)

☐ 2^5 (divide by 32)

☐ 2^2 (divide by 4)

☐ 2^6 (divide by 64)

☐ 2^3 (divide by 8)

☐ 2^7 (divide by 128)

Internal Oscillator Input

☐ Crystal

☐ Resistor

Low Voltage Inhibit

☐ Disable

☐ Enable

Customer Name _____

Address _____

City _____ State _____ Zip _____

Phone (____) _____ Extension _____

Contact Ms/Mr _____

Customer Part Number _____

Pattern Media

2708 PROM

2716 PROM

Paper Object Tape

Silent 700 Cassette

Card Deck

Tape of Card Deck

MDOS Disk File

(Note 2) _____

Notes: (2) Other media require prior factory approval.

Signature _____

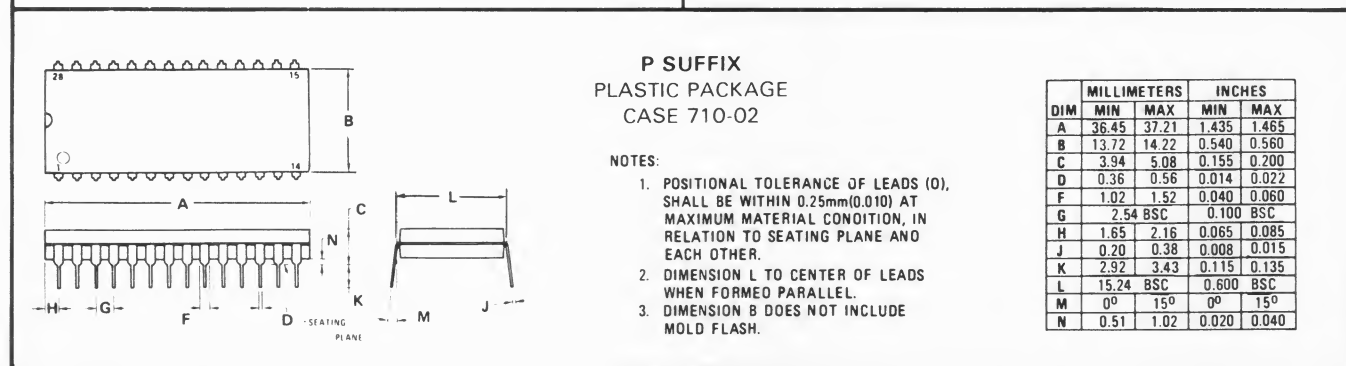
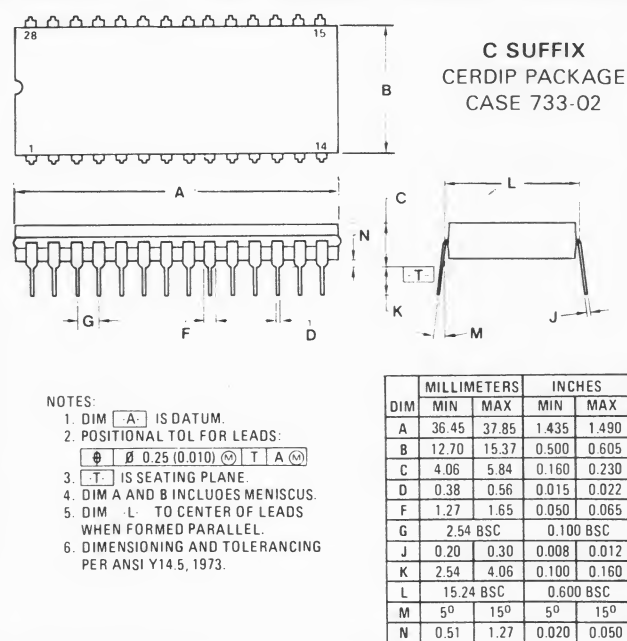
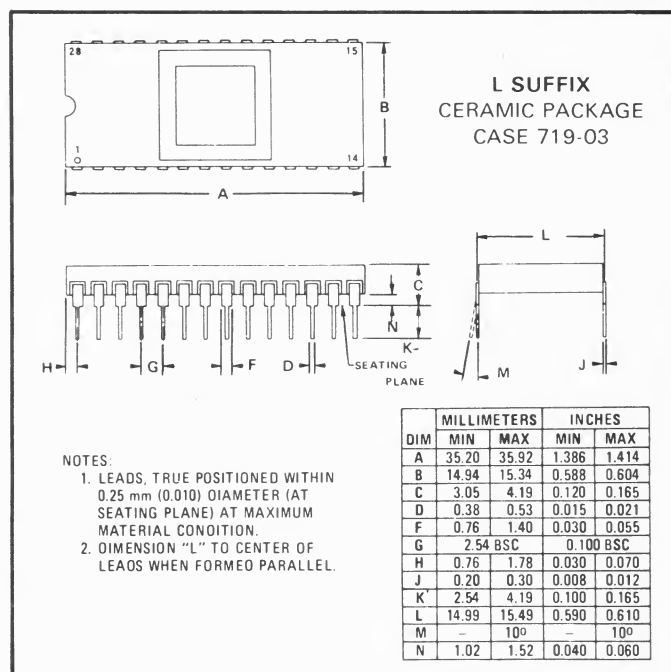
Title _____



TABLE 9 — THE GROWING M6805 FAMILY

M6805 Family System Configuration And Programming Features	MC6805P2	MC146805E2	MC6805R2	MC146805G2
Technology	NMOS	CMOS	NMOS	CMOS
Number of Pins	28	40	40	40
On-Chip RAM (bytes)	64	112	64	112
On-Chip User ROM (bytes)	1.1k	None	2k	2.2k
Expansion Bus	None	Yes	None	None
Bidirectional I/O Lines	20	16	32	32
I/O Options	None	None	A/D Converter	None
Software Compatibility	Similar to M6800		Similar M6800	
True Bit Manipulation	Yes	Yes	Yes	Yes
Instructions	59	61	59	61
Ten Addressing Modes	Yes	Yes	Yes	Yes

PACKAGE DIMENSIONS



Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.



MOTOROLA Semiconductor Products Inc.



MOTOROLA

Semiconductors

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

Advance Information

MICROPROCESSOR WITH CLOCK

The MC6808 is a monolithic 8-bit microprocessor that contains all the registers and accumulators of the present MC6800 plus an internal clock oscillator and driver on the same chip.

The MC6808 is completely software-compatible with the MC6800 as well as the entire M6800 family of parts. Hence the MC6808 is expandable to 65K words.

This very cost-effective MPU allows the designer to use the MC6808 in consumer as well as industrial applications without sacrificing industrial specifications.

- On-Chip Clock Circuit
- Software-Compatible with the MC6800
- Expandable to 65K words
- Standard TTL-Compatible Inputs and Outputs
- 8-Bit Word Size
- 16-Bit Memory Addressing
- Interrupt Capability

MC6808

MOS

(N-CHANNEL, SILICON-GATE,
DEPLETION LOAD)

MICROPROCESSOR WITH CLOCK

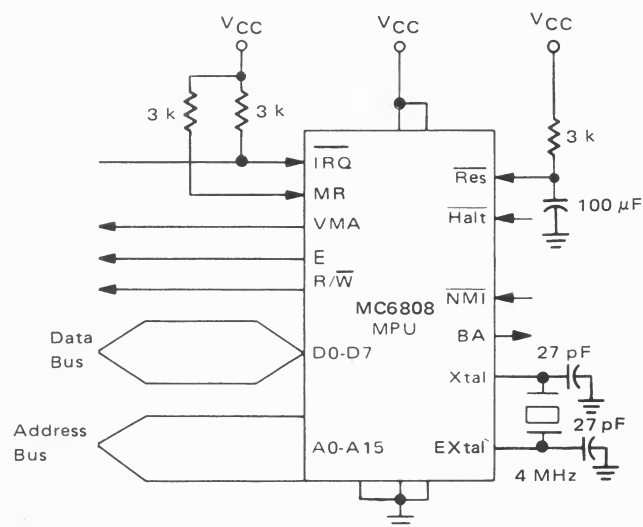


L SUFFIX
CERAMIC PACKAGE
CASE 715



P SUFFIX
PLASTIC PACKAGE
CASE 711

FIGURE 1 — TYPICAL MICROPROCESSOR INTERFACE



PIN ASSIGNMENT

1	V _{SS}	Reset	40
2	Halt	EXtal	39
3	MR	Xtal	38
4	IRQ	E	37
5	VMA	V _{SS}	36
6	NMI	V _{CC}	35
7	BA	R/W	34
8	V _{CC}	D0	33
9	A0	D1	32
10	A1	D2	31
11	A2	D3	30
12	A3	D4	29
13	A4	D5	28
14	A5	D6	27
15	A6	D7	26
16	A7	A15	25
17	A8	A14	24
18	A9	A13	23
19	A10	A12	22
20	A11	V _{SS}	21

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	Vdc
Input Voltage	V_{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C
Thermal Resistance	θ_{JA}	100 50	°C/W
Plastic			
Ceramic			

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

ELECTRICAL CHARACTERISTICS ($V_{CC} = 5.0 \text{ V} \pm 5\%$, $V_{SS} = 0$, $T_A = 0$ to 70°C unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Logic, $\overline{\text{EXtal}}$ $\overline{\text{Reset}}$	V_{IH}	$V_{SS} + 2.0$ $V_{SS} + 4.0$	— —	V_{CC} V_{CC}	Vdc
Input Low Voltage Logic, $\overline{\text{EXtal}}$ $\overline{\text{Reset}}$	V_{IL}	$V_{SS} - 0.3$ $V_{SS} - 0.3$	—	$V_{SS} + 0.8$ $V_{SS} + 2.3$	Vdc
Input Leakage Current ($V_{in} = 0$ to 5.25 V , $V_{CC} = \text{max}$)	I_{in}	—	1.0	2.5	μA
Output High Voltage ($I_{Load} = -205 \mu\text{A}$, $V_{CC} = \text{min}$) ($I_{Load} = -145 \mu\text{A}$, $V_{CC} = \text{min}$) ($I_{Load} = -100 \mu\text{A}$, $V_{CC} = \text{min}$)	V_{OH}	$V_{SS} + 2.4$ $V_{SS} + 2.4$ $V_{SS} + 2.4$	— — —	— — —	Vdc
Output Low Voltage ($I_{Load} = 1.6 \text{ mA}$, $V_{CC} = \text{min}$)	V_{OL}	—	—	$V_{SS} + 0.4$	Vdc
Power Dissipation	P_D^{**}	—	0.600	1.2	W
Capacitance # ($V_{in} = 0$, $T_A = 25^\circ\text{C}$, $f = 1.0 \text{ MHz}$)	C_{in}	—	10	12.5	pF
		—	6.5	10	
	C_{out}	—	—	12	pF
Frequency of Operation (Input Clock $\div 4$) (Crystal Frequency)	f f_{Xtal}	0.1 1.0	— —	1.0 4.0	MHz
Clock Timing					
Cycle Time	t_{cyc}	1.0	—	10	μs
Clock Pulse Width (measured at 2.4V) (measured at 0.4V)	$PW_{\phi Hs}$ $PW_{\phi L}$	450 450	—	4500 4500	ns
Fall Time (Measured between $V_{SS} + 0.4 \text{ V}$ and $V_{SS} + 2.4 \text{ V}$)	t_{ϕ}	—	—	25	ns

*Except $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$, which require $3 \text{ k}\Omega$ pullup load resistors for wire-OR capability at optimum operation. Does not include $\overline{\text{EXtal}}$ and Xtal , which are crystal inputs.

#Capacitances are periodically sampled rather than 100% tested.

READ/WRITE TIMING (Figures 2 through 6; Load Circuit of Figure 4.)

Characteristic	Symbol	Min	Typ	Max	Unit
Address Delay	t_{AD}	—	—	270	ns
Peripheral Read Access Time $t_{acc} = t_{ut} - (t_{AD} + t_{DSR})$; $t_{ut} = t_{cyc} - t_{\phi}$	t_{acc}	—	—	530	ns
Data Setup Time (Read)	t_{DSR}	100	—	—	ns
Input Data Hold Time	t_H	10	—	—	ns
Output Data Hold Time	t_H	30	—	—	ns
Address Hold Time (Address, R/\overline{W} , VMA)	t_{AH}	20	—	—	ns
Data Delay Time (Write)	t_{DDW}	—	165	225	ns
Processor Controls					
Processor Control Setup Time	t_{PCS}	200	—	—	ns
Processor Control Rise and Fall Time (Measured between 0.8 V and 2.0 V)	t_{PCr} , t_{PCf}	—	—	100	ns
Bus Available Delay Time	t_{BA}	—	—	250	ns



FIGURE 2 – READ DATA FROM MEMORY OR PERIPHERALS

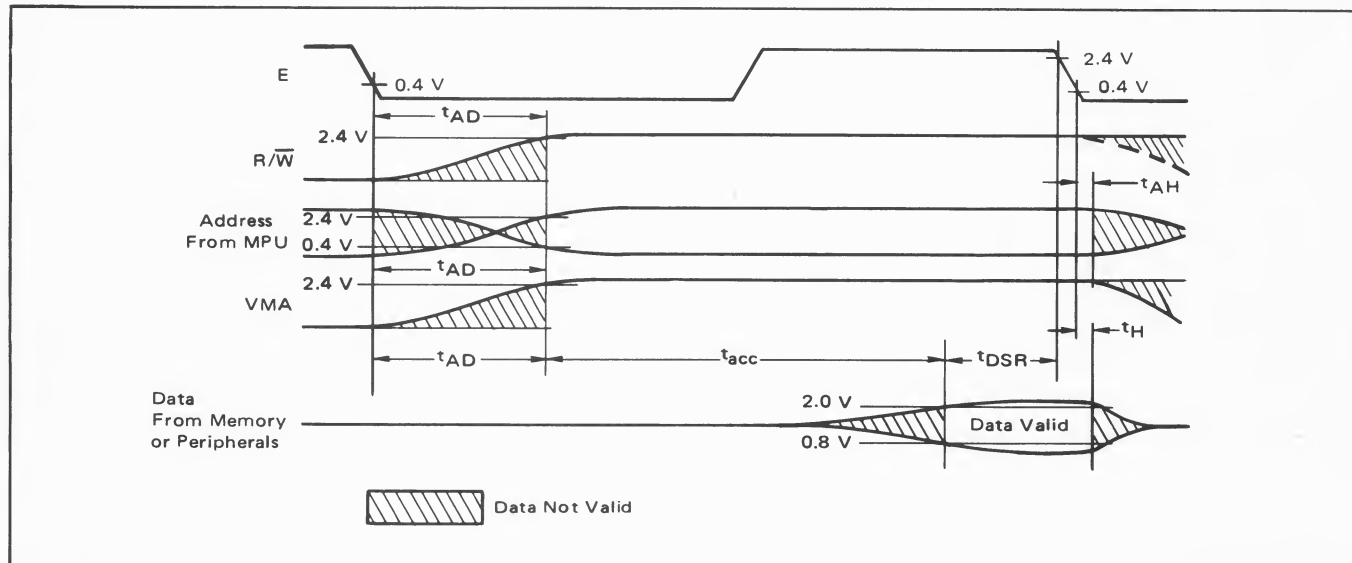


FIGURE 3 – WRITE DATA IN MEMORY OR PERIPHERALS

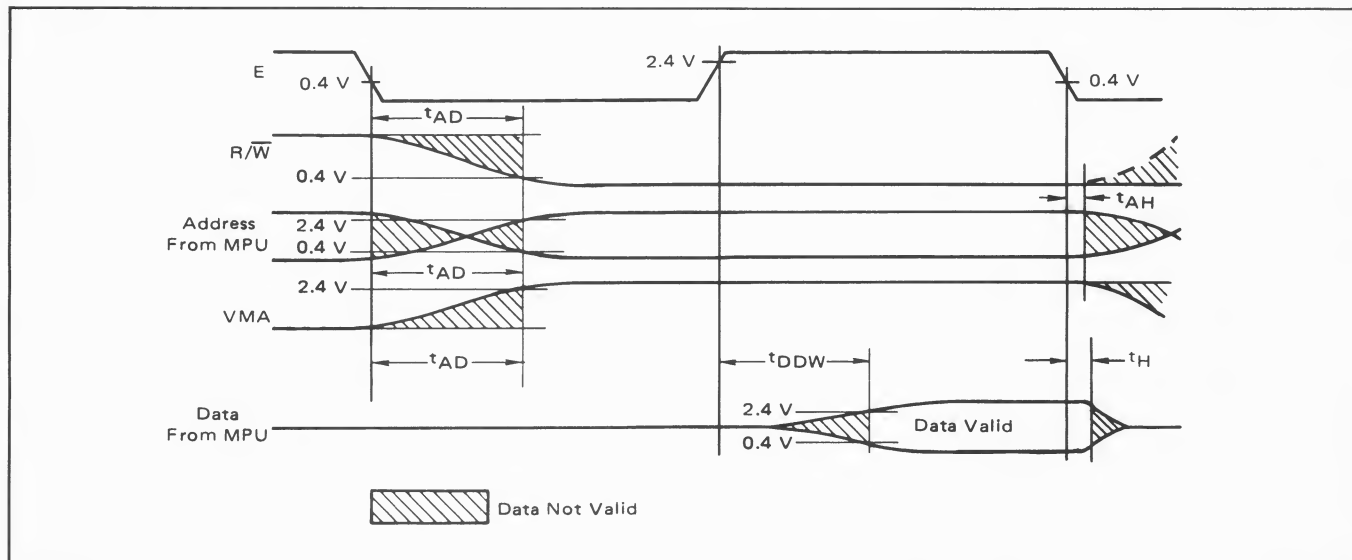


FIGURE 4 – BUS TIMING TEST LOAD

- $C = 130 \text{ pF}$ for D0-D7, E
 $= 90 \text{ pF}$ for A0-A15, R/W, and VMA
 $= 30 \text{ pF}$ for BA
 $R = 11.7 \text{ k}\Omega$ for D0-D7, E
 $= 16.5 \text{ k}\Omega$ for A0-A15, R/W, and VMA
 $= 24 \text{ k}\Omega$ for BA

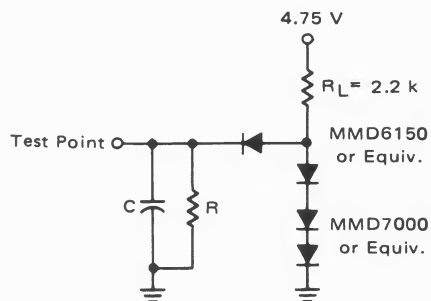


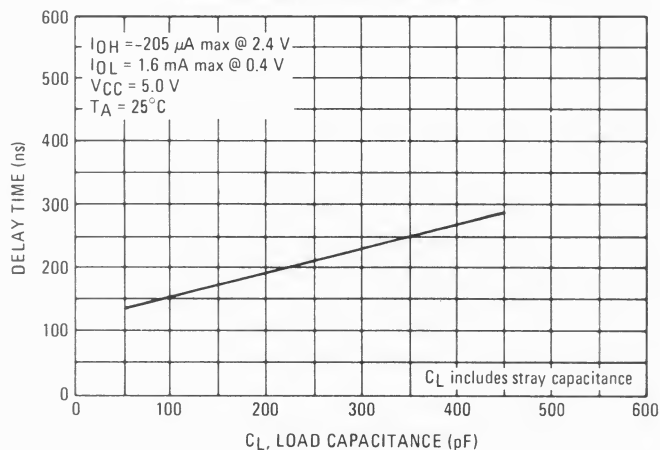
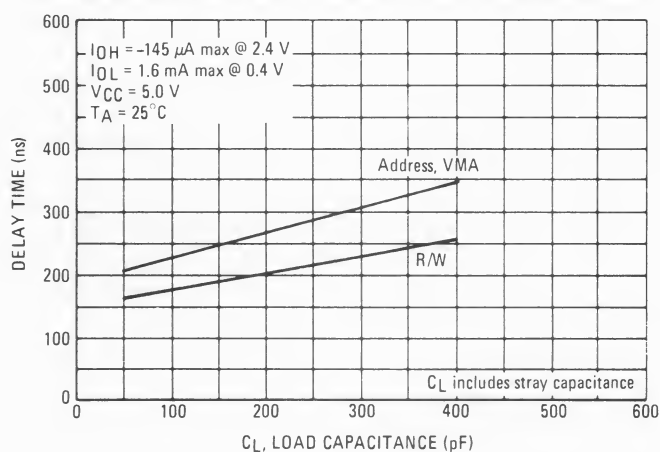
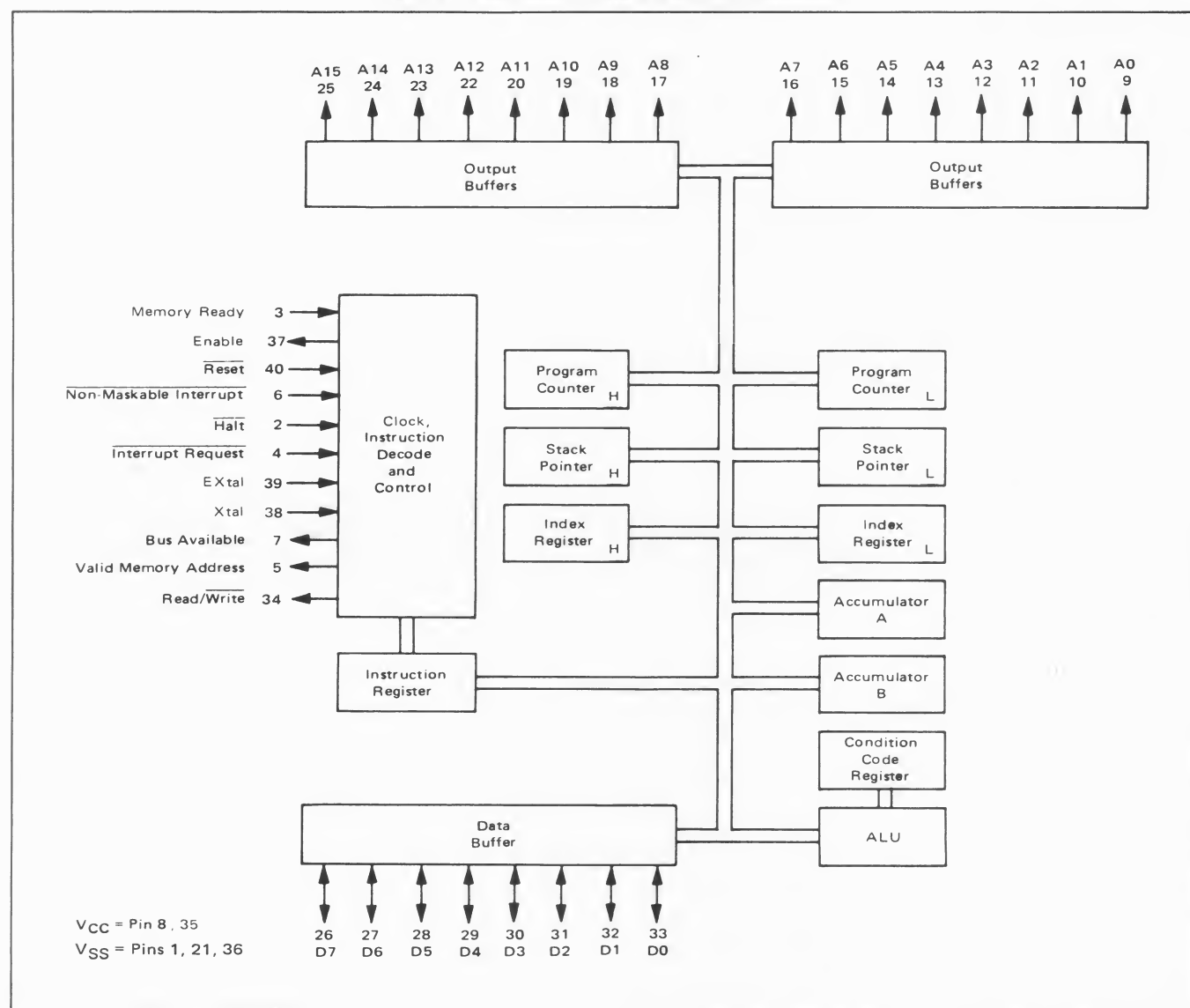
FIGURE 5 – TYPICAL DATA BUS OUTPUT DELAY
versus CAPACITIVE LOADINGFIGURE 6 – TYPICAL READ/WRITE, VMA, AND
ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING

FIGURE 7 – MC6808 EXPANDED BLOCK DIAGRAM



MPU REGISTERS

A general block diagram of the MC6808 is shown in Figure 7. As shown, the number and configuration of the registers are the same as for the MC6800.

The MPU has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 8).

Program Counter — The program counter is a two byte (16-bits) register that points to the current program address.

Stack Pointer — The stack pointer is a two byte register that contains the address of the next available location in an external push-down/pop-up stack. This stack is normally a random access Read/Write memory that may have any location (address) that is convenient. In those applications that require storage of information in the stack when power is lost, the stack must be non-volatile.

Index Register — The index register is a two byte register that is used to store data or a sixteen bit memory address for the Indexed mode of memory addressing.

Accumulators — The MPU contains two 8-bit accumulators that are used to hold operands and results from an arithmetic logic unit (ALU).

Condition Code Register — The condition code register indicates the results of an Arithmetic Logic Unit operation: Negative (N), Zero (Z), Overflow (V), Carry from bit 7 (C), and half carry from bit 3 (H). These bits of the Condition Code Register are used as testable conditions for the conditional branch instructions. Bit 4 is the interrupt mask bit (I). The used bits of the Condition Code Register (b6 and b7) are ones.

Figure 9 shows the order of saving the microprocessor status within the stack.

FIGURE 8 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT

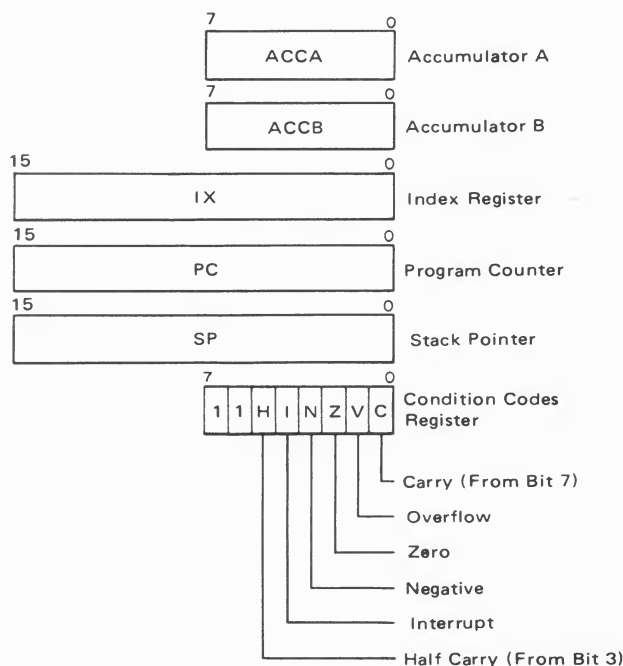
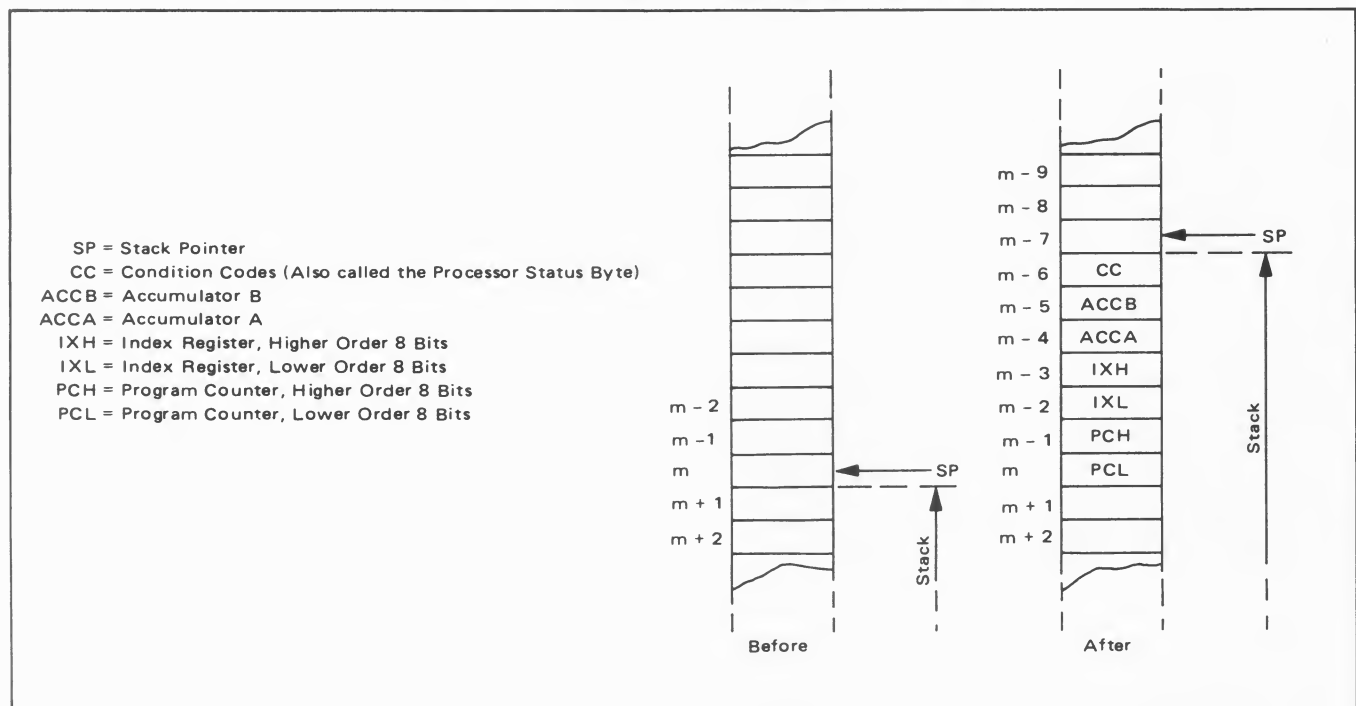


FIGURE 9 — SAVING THE STATUS OF THE MICROPROCESSOR IN THE STACK



MC6808 MPU SIGNAL DESCRIPTION

Proper operation of the MPU requires that certain control and timing signals be provided to accomplish specific functions and that other signal lines be monitored to determine the state of the processor. These control and timing signals for the MC6808 are similar to those of the MC6800 except that TSC, DBE, $\phi 1$, $\phi 2$ input, and two unused pins have been eliminated, and the following signal and timing lines have been added:

- Crystal Connections EXtal and Xtal
- Memory Ready (MR)
- Enable $\phi 2$ Output (E)

The following is a summary of the MC6808 MPU signals:

Address Bus (A0-A15) — Sixteen pins are used for the address bus. The outputs are capable of driving one standard TTL load and 90 pF.

Data Bus (D0-D7) — Eight pins are used for the data bus. It is bidirectional, transferring data to and from the memory and peripheral devices. It also has three-state output buffers capable of driving one standard TTL load and 130 pF.

Halt — When this input is in the low state, all activity in the machine will be halted. This input is level sensitive.

In the halt mode, the machine will stop at the end of an instruction, Bus Available will be at a high state, Valid Memory Address will be at a low state, and all other three-state lines will be in the three-state mode. The address bus will display the address of the next instruction.

To insure single instruction operation, transition of the Halt line must not occur during the last 200 ns of E and the Halt line must go high for one Clock cycle.

Read/Write (R/\bar{W}) — This TTL compatible output signals the peripherals and memory devices whether the MPU is in a Read (high) or Write (low) state. The normal standby state of this signal is Read (high). When the processor is halted, it will be in the logical one state. This output is capable of driving one standard TTL load and 90 pF.

Valid Memory Address (VMA) — This output indicates to peripheral devices that there is a valid address on the address bus. In normal operation, this signal should be utilized for enabling peripheral interfaces such as the PIA and ACIA. This signal is not three-state. One standard TTL load and 90 pF may be directly driven by this active high signal.



Bus Available (BA) — The Bus Available signal will normally be in the low state; when activated, it will go to the high state indicating that the microprocessor has stopped and that the address bus is available but not in three-state. This will occur if the $\overline{\text{Halt}}$ line is in the low state or the processor is in the WAIT state as a result of the execution of a WAIT instruction. At such time, all three-state output drivers will go to their off state and other outputs to their normally inactive level. The processor is removed from the WAIT state by the occurrence of a maskable (mask bit $I = 0$) or nonmaskable interrupt. This output is capable of driving one standard TTL load and 30 pF.

Interrupt Request ($\overline{\text{IRQ}}$) — This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further interrupts may occur. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations

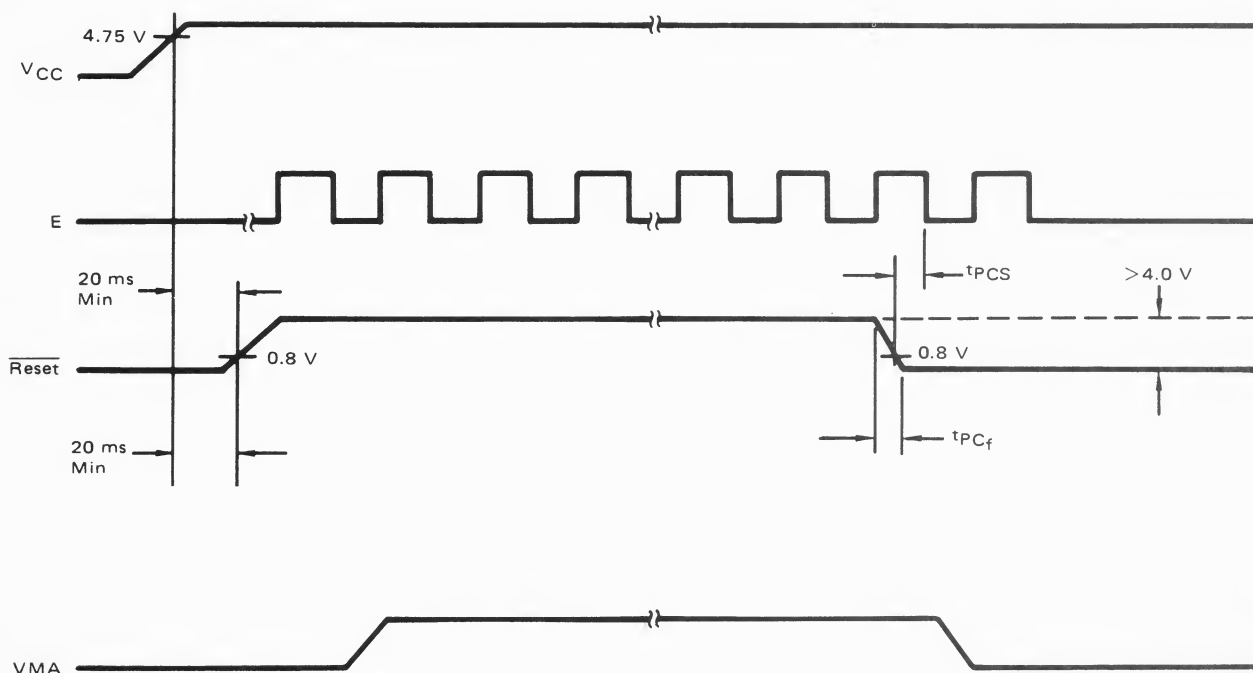
FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine in memory.

The $\overline{\text{Halt}}$ line must be in the high state for interrupts to be serviced. Interrupts will be latched internally while $\overline{\text{Halt}}$ is low.

The $\overline{\text{IRQ}}$ has a high impedance pullup device internal to the chip; however a 3 k Ω external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

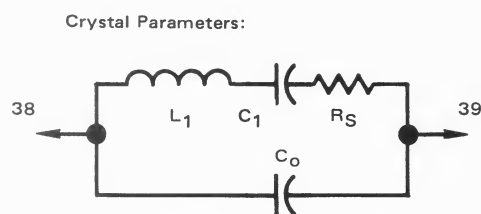
Reset — This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. When this line is low, the MPU is inactive and the information in the registers will be lost. If a high level is detected on the input, this will signal the MPU to begin the restart sequence. This will start execution of a routine to initialize the processor from its reset condition. All the higher order address lines will be forced high. For the restart, the last two (FFFE, FFFF) locations in memory will be used to load the program that is addressed by the program counter. During the restart routine, the interrupt mask bit is set and must be reset before the MPU can be interrupted by $\overline{\text{IRQ}}$. Power-up and reset timing sequences are shown in Figure 10.

FIGURE 10 — POWER-UP AND RESET TIMING



EXtal and Xtal — The MC 6808 has an internal oscillator that may be crystal controlled. These connections are for a parallel resonant fundamental crystal. (AT cut.) A divide-by-four circuit has been added to the MC6808 so that a 4 MHz crystal may be used in lieu of a 1 MHz crystal for a more cost-effective system. Pin 39 of the MC6808 may be driven externally by a TTL input signal if a separate clock is required. Pin 38 is to be left open in this mode. Crystal parameters to be specified are in Figure 11.

FIGURE 11—CRYSTAL PARAMETERS



AT — Cut Parallel Resonance Crystal
 $C_0 = 7 \text{ pF Max.}$
 $\text{FREQ} = 4.0 \text{ MHz @ } C_L = 24 \text{ pF}$
 $R_S = 50 \text{ ohms Max.}$
 Frequency Tolerance — $\pm 5\%$ to $\pm 0.02\%$
 The best E output "Worst Case Design"
 tolerance is $\pm 0.05\%$ (500 ppm) using
 $A \pm 0.02\%$ crystal.

Non-Maskable Interrupt (NMI) — A low-going edge on this input requests that a non-mask-interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the NMI signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

The index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations FFFC and FFFD. An address loaded at these locations caused the MPU to branch to a non-maskable interrupt routine in memory.

NMI has a high impedance pullup resistor internal to the chip; however a $3 \text{ k}\Omega$ external resistor to V_{CC} should be used for wire-OR and optimum control of interrupts.

Inputs IRQ and NMI are hardware interrupt lines that are sampled when E is high and will start the interrupt routine on a low E following the completion of an instruction.

Figure 12 is a flow chart describing the major decision paths and interrupt vectors of the microprocessor. Table 1 gives the memory map for interrupt vectors.

Memory Ready (MR) — MR is a TTL compatible input control signal which allows stretching of E. When MR is high, E will be in normal operation. When MR is low, E may be stretched integral multiples of half periods, thus allowing interface to slow memories. Memory Ready timing is shown in Figure 13.

TABLE 1 — MEMORY MAP FOR INTERRUPT VECTORS

Vector		Description
MS	LS	
FFFE	FFFF	Restart
FFFC	FFFD	Non-maskable Interrupt
FFFA	FFFB	Software Interrupt
FFF8	FFF9	Interrupt Request



FIGURE 12 – MPU FLOW CHART

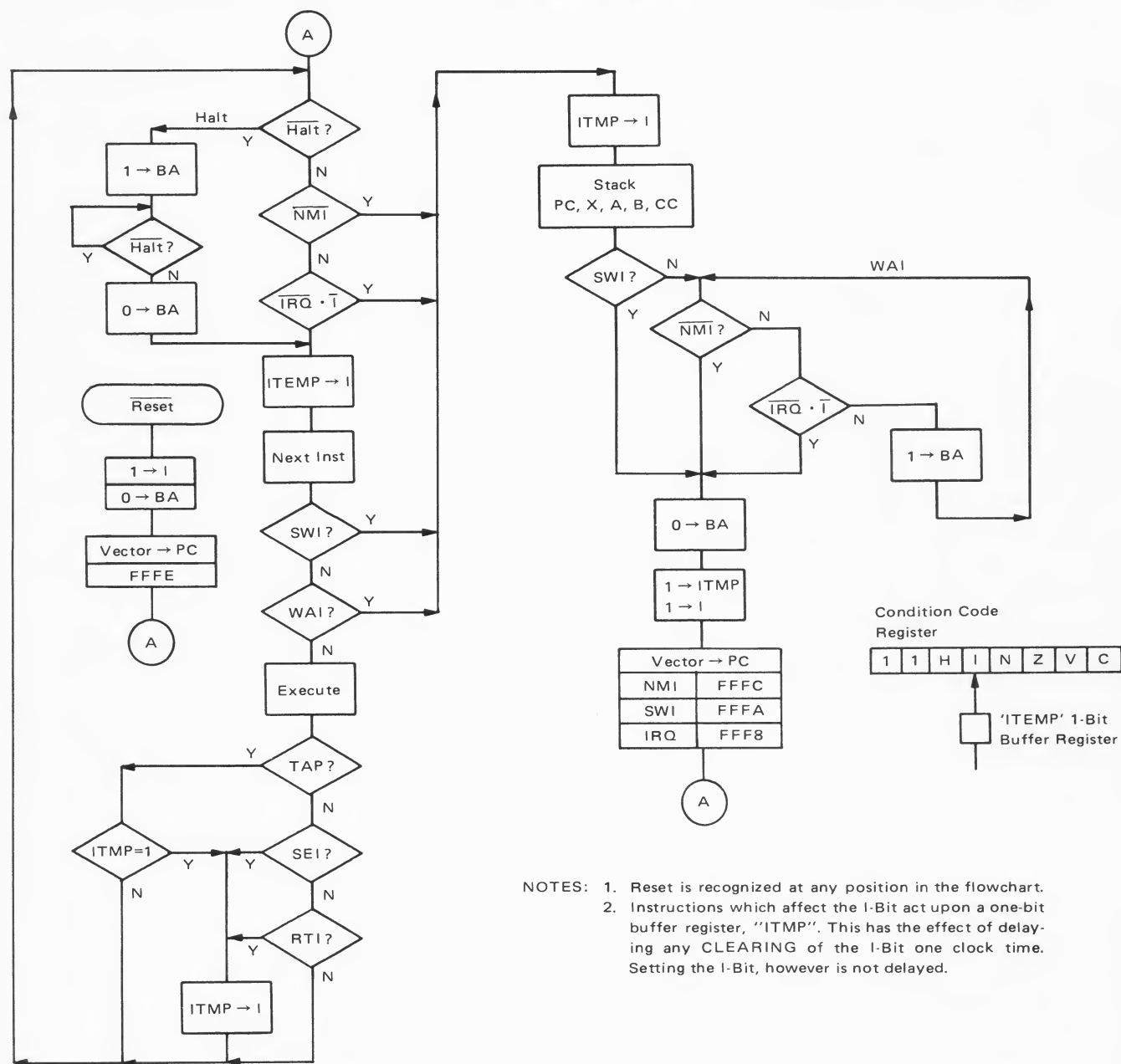
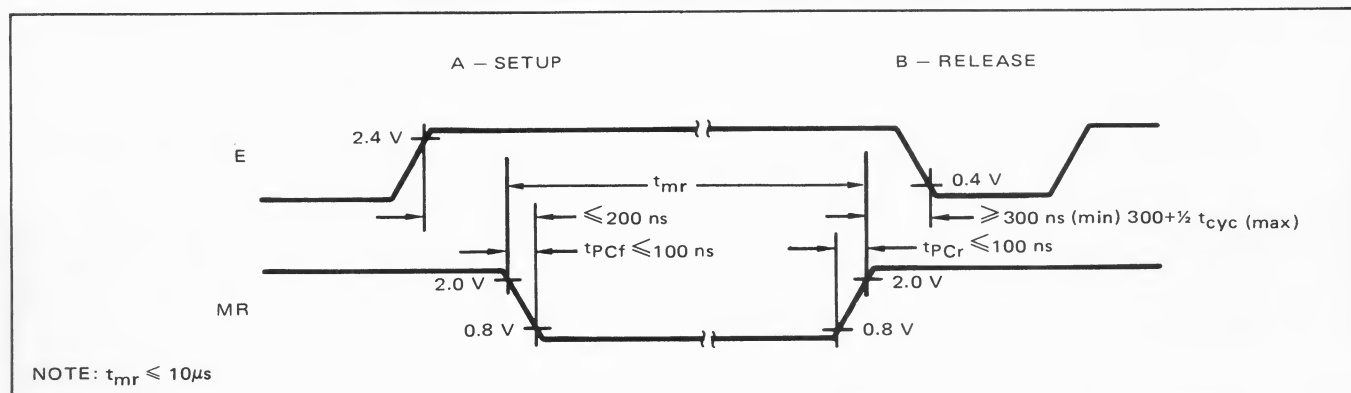


FIGURE 13 – MEMORY READY CONTROL FUNCTION



MPU INSTRUCTION SET

The MC6808 has a set of 72 different instructions. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, conditional or unconditional branch, interrupt and stack manipulation instructions (Tables 2 thru 6). This instruction set is the same as that for the MC6800.

MPU ADDRESSING MODES

The MC6808 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 7 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 1 MHz, these times would be microseconds.

Accumulator (ACCX) Addressing — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

Immediate Addressing — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MPU addresses this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

Direct Addressing — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

Extended Addressing — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

Indexed Addressing — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

Implied Addressing — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

Relative Addressing — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.

TABLE 2 — MICROPROCESSOR INSTRUCTION SET — ALPHABETIC SEQUENCE

ABA	Add Accumulators	CLR	Clear	PUL	Pull Data
ADC	Add with Carry	CLV	Clear Overflow	ROL	Rotate Left
ADD	Add	CMP	Compare	ROR	Rotate Right
AND	Logical And	COM	Complement	RTI	Return from Interrupt
ASL	Arithmetic Shift Left	CPX	Compare Index Register	RTS	Return from Subroutine
ASR	Arithmetic Shift Right	DAA	Decimal Adjust	SBA	Subtract Accumulators
BCC	Branch if Carry Clear	DEC	Decrement	SBC	Subtract with Carry
BCS	Branch if Carry Set	DES	Decrement Stack Pointer	SEC	Set Carry
BEQ	Branch if Equal to Zero	DEX	Decrement Index Register	SEI	Set Interrupt Mask
BGE	Branch if Greater or Equal Zero	EOR	Exclusive OR	SEV	Set Overflow
BGT	Branch if Greater than Zero	INC	Increment	STA	Store Accumulator
BHI	Branch if Higher	INS	Increment Stack Pointer	STS	Store Stack Register
BIT	Bit Test	INX	Increment Index Register	STX	Store Index Register
BLE	Branch if Less or Equal	JMP	Jump	SUB	Subtract
BLS	Branch if Lower or Same	JSR	Jump to Subroutine	SWI	Software Interrupt
BLT	Branch if Less than Zero	LDA	Load Accumulator	TAB	Transfer Accumulators
BMI	Branch if Minus	LDS	Load Stack Pointer	TAP	Transfer Accumulators to Condition Code Reg.
BNE	Branch if Not Equal to Zero	LDX	Load Index Register	TBA	Transfer Accumulators
BPL	Branch if Plus	LSR	Logical Shift Right	TPA	Transfer Condition Code Reg. to Accumulator
BRA	Branch Always	NEG	Negate	TST	Test
BSR	Branch to Subroutine	NOP	No Operation	TSX	Transfer Stack Pointer to Index Register
BVC	Branch if Overflow Clear	ORA	Inclusive OR Accumulator	TXS	Transfer Index Register to Stack Pointer
BVS	Branch if Overflow Set	PSH	Push Data	WAI	Wait for Interrupt
CBA	Compare Accumulators				
CLC	Clear Carry				
CLI	Clear Interrupt Mask				



TABLE 3 — ACCUMULATOR AND MEMORY INSTRUCTIONS

ADDRESSING MODES										BOOLEAN/ARITHMETIC OPERATION										COND. CODE REG.																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
OPERATIONS		MNEMONIC		IMMED		DIRECT		INDEX		EXTND		IMPLIED		(All register labels refer to contents)																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													
				OP	~ =	OP	~ =	OP	~ =	OP	~ =	OP	~ =											OP	~ =	H	I	N	Z	V	C																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
Add	ADDA	38	2 2	98	3 2	A8	5 2	B8	4 3					A + M ← A																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													

LEGEND:

OP Operation Code (Hexadecimal);
 ~ Number of MPU Cycles;
 = Number of Program Bytes;
 + Arithmetic Plus;
 - Arithmetic Minus;
 • Boolean AND;
 Msp Contents of memory location pointed to by Stack Pointer;

+ Boolean Inclusive OR;
 ⊖ Boolean Exclusive OR;
 M Complement of M;
 → Transfer Into;
 0 Bit = Zero;
 00 Byte = Zero;

CONDITION CODE SYMBOLS:

H Half-carry from bit 3;
 I Interrupt mask;
 N Negative (sign bit)
 Z Zero (byte)
 V Overflow, 2's complement
 C Carry from bit 7
 R Reset Always
 S Set Always
 : Test and set if true, cleared otherwise
 • Not Affected

Note — Accumulator addressing mode instructions are included in the column for IMPLIED addressing



TABLE 4 – INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

POINTER OPERATIONS	MNEMONIC																COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			IMPLIED			BOOLEAN/ARITHMETIC OPERATION					
		OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=	H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3						7	8		
Decrement Index Reg	DEX													09	4	1						
Decrement Stack Pntr	DES													34	4	1						
Increment Index Reg	INX													08	4	1						
Increment Stack Pntr	INS													31	4	1						
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3						9		R	
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3						9		R	
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3						9		R	
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3						9		R	
Idx Reg → Stack Pntr	TXS													35	4	1						
Stack Pntr → Idx Reg	TSX													30	4	1						
BOOLEAN/ARITHMETIC OPERATION																						
$X_H - M, X_L - (M + 1)$																						
$X - 1 \rightarrow X$																						
$SP - 1 \rightarrow SP$																						
$X + 1 \rightarrow X$																						
$SP + 1 \rightarrow SP$																						
$M \rightarrow X_H, (M + 1) \rightarrow X_L$																						
$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$																						
$X_H \rightarrow M, X_L \rightarrow (M + 1)$																						
$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$																						
$X - 1 \rightarrow SP$																						
$SP + 1 \rightarrow X$																						

TABLE 5 – JUMP AND BRANCH INSTRUCTIONS

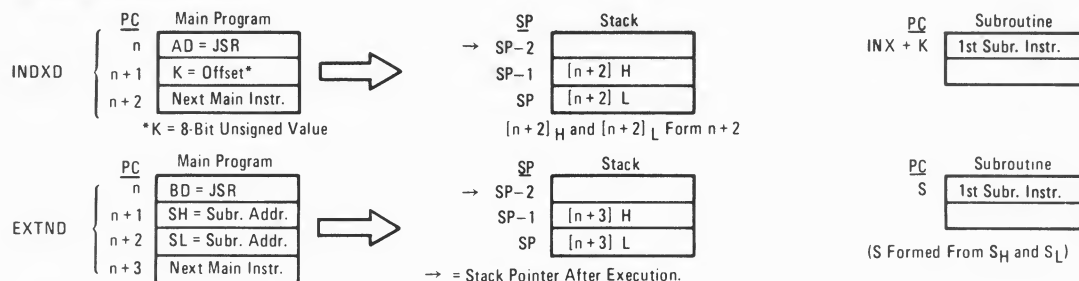
														COND. CODE REG.							
OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			IMPLIED			BRANCH TEST	5	4	3	2	1	0	
		OP	~	=	OP	~	=	OP	~	=	OP	~	=		H	I	N	Z	V	C	
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•	
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•	
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•	
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•	
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	•	•	•	•	•	•	
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	•	•	•	•	•	•	
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•	
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	•	•	•	•	•	•	
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•	
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	•	•	•	•	•	•	
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•	
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•	
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•	
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•	
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•	
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•	
Jump	JMP				6E	4	2	7E	3	3				} See Special Operations	•	•	•	•	•	•	
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•	
Return From Interrupt	RTI										3B	10	1	} See Special Operations	•	•	•	•	•	•	
Return From Subroutine	RTS										39	5	1		•	•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1		•	•	•	•	•	•	•
Wait for Interrupt *	WAI										3E	9	1		•	11	•	•	•	•	

*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

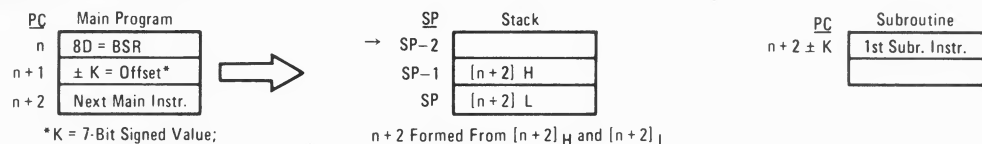


SPECIAL OPERATIONS

JSR, JUMP TO SUBROUTINE:



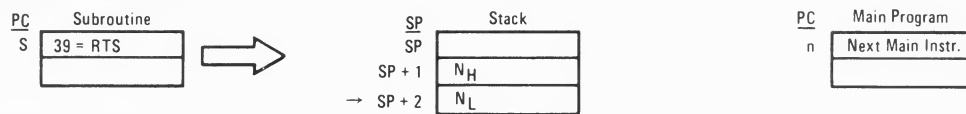
BSR, BRANCH TO SUBROUTINE:



JMP, JUMP:



RTS, RETURN FROM SUBROUTINE:



RTI, RETURN FROM INTERRUPT:

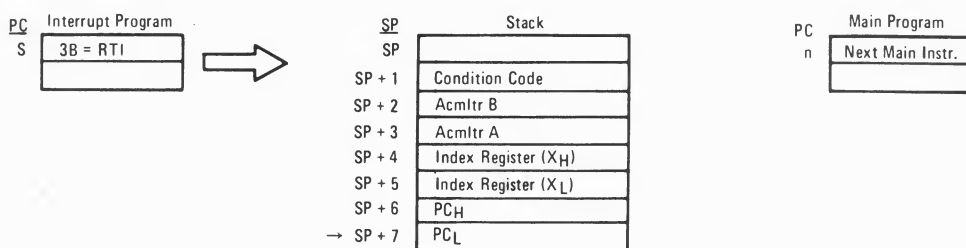


TABLE 6 – CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

OPERATIONS	MNEMONIC	IMPLIED			BOOLEAN OPERATION	COND. CODE REG.						
		OP	~	=		5	4	3	2	1	0	
						H	I	N	Z	V	C	
Clear Carry	CLC	0C	2	1	0 → C	•	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	0 → I	•	R	•	•	•	•	•
Clear Overflow	CLV	0A	2	1	0 → V	•	•	•	•	R	•	•
Set Carry	SEC	0D	2	1	1 → C	•	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	1 → I	•	S	•	•	•	•	•
Set Overflow	SEV	0B	2	1	1 → V	•	•	•	•	•	S	•
Accmltr A → CCR	TAP	06	2	1	A → CCR	12						
CCR → Accmltr A	TPA	07	2	1	CCR → A							

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- | | |
|---------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| 1 (Bit V) Test: Result = 10000000? | 7 (Bit N) Test: Sign bit of most significant (MS) byte = 1? |
| 2 (Bit C) Test: Result ≠ 00000000? | 8 (Bit V) Test: 2's complement overflow from subtraction of MS bytes? |
| 3 (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.) | 9 (Bit N) Test: Result less than zero? (Bit 15 = 1) |
| 4 (Bit V) Test: Operand = 10000000 prior to execution? | 10 (All) Load Condition Code Register from Stack. (See Special Operations) |
| 5 (Bit V) Test: Operand = 01111111 prior to execution? | 11 (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state. |
| 6 (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred. | 12 (All) Set according to the contents of Accumulator A. |



TABLE 7 — INSTRUCTION ADDRESSING MODES AND ASSOCIATED EXECUTION TIMES
(Times in Machine Cycles)

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied	Relative		(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied
ABA		•	•	•	•	•	2	•	INC	2	•	•	•	6	7	•
ADC	x	•	2	3	4	5	•	•	INS	•	•	•	•	•	•	4
ADD	x	•	2	3	4	5	•	•	INX	•	•	•	•	•	•	4
AND	x	•	2	3	4	5	•	•	JMP	•	•	•	3	4	•	•
ASL		2	•	•	6	7	•	•	JSR	•	•	•	9	8	•	•
ASR		2	•	•	6	7	•	•	LDA	x	2	3	4	5	•	•
BCC		•	•	•	•	•	•	4	LDS	•	3	4	5	6	•	•
BCS		•	•	•	•	•	•	4	LDX	•	3	4	5	6	•	•
BEA		•	•	•	•	•	•	4	LSR	2	•	•	6	7	•	•
BGE		•	•	•	•	•	•	4	NEG	2	•	•	6	7	•	•
BGT		•	•	•	•	•	•	4	NOP	•	•	•	•	•	•	2
BHI		•	•	•	•	•	•	4	ORA	x	2	3	4	5	•	•
BIT	x	•	2	3	4	5	•	•	PSH	•	•	•	•	•	•	4
BLE		•	•	•	•	•	•	4	PUL	•	•	•	•	•	•	4
BLS		•	•	•	•	•	•	4	ROL	2	•	•	6	7	•	•
BLT		•	•	•	•	•	•	4	ROR	2	•	•	6	7	•	•
BMI		•	•	•	•	•	•	4	RTI	•	•	•	•	•	•	10
BNE		•	•	•	•	•	•	4	RTS	•	•	•	•	•	•	5
BPL		•	•	•	•	•	•	4	SBA	•	•	•	•	•	•	2
BRA		•	•	•	•	•	•	4	SBC	x	•	2	3	4	5	•
BSR		•	•	•	•	•	•	8	SEC	•	•	•	•	•	•	2
BVC		•	•	•	•	•	•	4	SEI	•	•	•	•	•	•	2
BVS		•	•	•	•	•	•	4	SEV	•	•	•	•	•	•	2
CBA		•	•	•	•	•	2	•	STA	x	•	•	4	5	6	•
CLC		•	•	•	•	•	2	•	STS	•	•	5	6	7	•	•
CLI		•	•	•	•	•	2	•	STX	•	•	5	6	7	•	•
CLR		2	•	•	6	7	•	•	SUB	x	•	2	3	4	5	•
CLV		•	•	•	•	•	2	•	SWI	•	•	•	•	•	•	12
CMP	x	•	2	3	4	5	•	•	TAB	•	•	•	•	•	•	2
COM		2	•	•	6	7	•	•	TAP	•	•	•	•	•	•	2
CPX		•	3	4	5	6	•	•	TBA	•	•	•	•	•	•	2
DAA		•	•	•	•	•	2	•	TPA	•	•	•	•	•	•	2
DEC		2	•	•	6	7	•	•	TST	2	•	•	6	7	•	•
DES		•	•	•	•	•	4	•	TSX	•	•	•	•	•	•	4
DEX		•	•	•	•	•	4	•	TSX	•	•	•	•	•	•	4
EOR	x	•	2	3	4	5	•	•	WAI	•	•	•	•	•	•	9

NOTE: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.



SUMMARY OF CYCLE BY CYCLE OPERATION

Table 8 provides a detailed description of the information present on the Address Bus, Data Bus, Valid Memory Address line (VMA), and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expected results during debug of both software and hard-

ware as the control program is executed. The information is categorized in groups according to Addressing Mode and Number of Cycles per instruction. (In general, instructions with the same Addressing Mode and Number of Cycles execute in the same manner; exceptions are indicated in the table.)

TABLE 8 — OPERATION SUMMARY

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
IMMEDIATE						
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1 2	1 1	Op Code Address Op Code Address + 1	1 1	Op Code Operand Data
CPX LDS LDX	3	1 2 3	1 1 1	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte)
DIRECT						
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1 2 3	1 1 1	Op Code Address Op Code Address + 1 Address of Operand	1 1 1	Op Code Address of Operand Operand Data
CPX LDS LDX	4	1 2 3 4	1 1 1 1	Op Code Address Op Code Address + 1 Address of Operand Operand Address + 1	1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte)
STA	4	1 2 3 4	1 1 0 1	Op Code Address Op Code Address + 1 Destination Address Destination Address	1 1 1 0	Op Code Destination Address Irrelevant Data (Note 1) Data from Accumulator
STS STX	5	1 2 3 4 5	1 1 0 1 1	Op Code Address Op Code Address + 1 Address of Operand Address of Operand Address of Operand + 1	1 1 1 0 0	Op Code Address of Operand Irrelevant Data (Note 1) Register Data (High Order Byte) Register Data (Low Order Byte)
INDEXED						
JMP	4	1 2 3 4	1 1 0 0	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry)	1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	5	1 2 3 4 5	1 1 0 0 1	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry) Index Register Plus Offset	1 1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1) Operand Data
CPX LDS LDX	6	1 2 3 4 5 6	1 1 0 0 1 1	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry) Index Register Plus Offset Index Register Plus Offset + 1	1 1 1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1) Operand Data (High Order Byte) Operand Data (Low Order Byte)



TABLE 8 – OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INDEXED (Continued)						
STA	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	1	Index Register Plus Offset	1	Current Operand Data
		6	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		7	1/0 (Note 3)	Index Register Plus Offset	0	New Operand Data (Note 3)
STS STX	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data (High Order Byte)
		7	1	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
JSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer — 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer — 2	1	Irrelevant Data (Note 1)
		7	0	Index Register	1	Irrelevant Data (Note 1)
		8	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
EXTENDED						
JMP	3	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Jump Address (High Order Byte)
		3	1	Op Code Address + 2	1	Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data
CPX LDS LDX	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data (High Order Byte)
		5	1	Address of Operand + 1	1	Operand Data (Low Order Byte)
STA A STA B	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Destination Address (High Order Byte)
		3	1	Op Code Address + 2	1	Destination Address (Low Order Byte)
		4	0	Operand Destination Address	1	Irrelevant Data (Note 1)
		5	1	Operand Destination Address	0	Data from Accumulator
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Current Operand Data
		5	0	Address of Operand	1	Irrelevant Data (Note 1)
		6	1/0 (Note 3)	Address of Operand	0	New Operand Data (Note 3)

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
EXTENDED (Continued)						
STS STX	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	0	Address of Operand	1	Irrelevant Data (Note 1)
		5	1	Address of Operand	0	Operand Data (High Order Byte)
		6	1	Address of Operand + 1	0	Operand Data (Low Order Byte)
JSR	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Subroutine (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
		4	1	Subroutine Starting Address	1	Op Code of Next Instruction
		5	1	Stack Pointer	0	Return Address (Low Order Byte)
		6	1	Stack Pointer — 1	0	Return Address (High Order Byte)
		7	0	Stack Pointer — 2	1	Irrelevant Data (Note 1)
		8	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		9	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
INHERENT						
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1	1	Op Code Address	1	Op Code
2		1	Op Code Address + 1	1	Op Code of Next Instruction	
DES DEX INS INX	4	1	1	Op Code Address	1	Op Code
2		1	Op Code Address + 1	1	Op Code of Next Instruction	
3		0	Previous Register Contents	1	Irrelevant Data (Note 1)	
4		0	New Register Contents	1	Irrelevant Data (Note 1)	
PSH	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	1	Stack Pointer	0	Accumulator Data
		4	0	Stack Pointer — 1	1	Accumulator Data
PUL	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Operand Data from Stack
TSX	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	0	New Index Register	1	Irrelevant Data (Note 1)
TXS	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	0	Index Register	1	Irrelevant Data
		4	0	New Stack Pointer	1	Irrelevant Data
RTS	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Address of Next Instruction (High Order Byte)
		5	1	Stack Pointer + 2	1	Address of Next Instruction (Low Order Byte)



TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INHERENT (Continued)						
WAI	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer – 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer – 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer – 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer – 4	0	Contents of Accumulator A
		8	1	Stack Pointer – 5	0	Contents of Accumulator B
		9	1	Stack Pointer – 6 (Note 4)	1	Contents of Cond. Code Register
RTI	10	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Contents of Cond. Code Register from Stack
		5	1	Stack Pointer + 2	1	Contents of Accumulator B from Stack
		6	1	Stack Pointer + 3	1	Contents of Accumulator A from Stack
		7	1	Stack Pointer + 4	1	Index Register from Stack (High Order Byte)
		8	1	Stack Pointer + 5	1	Index Register from Stack (Low Order Byte)
		9	1	Stack Pointer + 6	1	Next Instruction Address from Stack (High Order Byte)
			10	1	Stack Pointer + 7	1
SWI	12	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 1)
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer – 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer – 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer – 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer – 4	0	Contents of Accumulator A
		8	1	Stack Pointer – 5	0	Contents of Accumulator B
		9	1	Stack Pointer – 6	0	Contents of Cond. Code Register
		10	0	Stack Pointer – 7	1	Irrelevant Data (Note 1)
		11	1	Vector Address FFFA (Hex)	1	Address of Subroutine (High Order Byte)
		12	1	Vector Address FFFB (Hex)	1	Address of Subroutine (Low Order Byte)
RELATIVE						
BCC BHI BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMI BVS	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		4	0	Branch Address	1	Irrelevant Data (Note 1)
BSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer – 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer – 2	1	Irrelevant Data (Note 1)
		7	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		8	0	Subroutine Address	1	Irrelevant Data (Note 1, Note 5)

Note 1. If device which is addressed during this cycle uses VMA, then the Data Bus will go to the high impedance three-state condition. Depending on bus capacitance, data from the previous cycle may be retained on the Data Bus.

Note 2. Data is ignored by the MPU.

Note 3. For TST, VMA = 0 and Operand data does not change.

Note 4. While the MPU is waiting for the interrupt, Bus Available will go high, VMA is low.

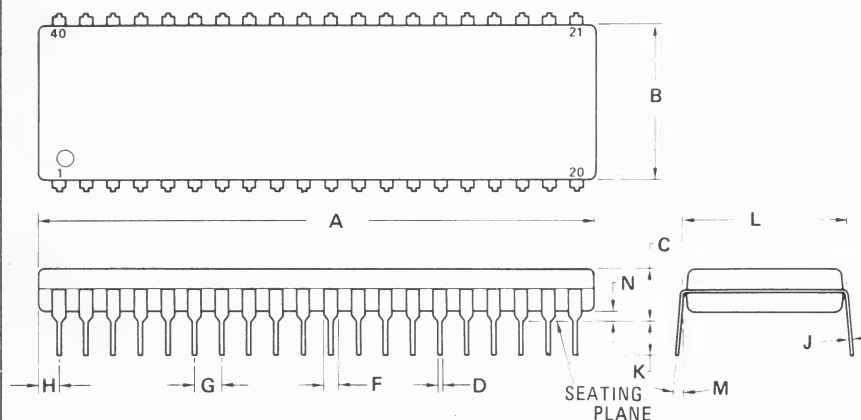
Note 5. MS Byte = MS Byte of BSR instruction address, LS Byte = LS Byte of subroutine address.



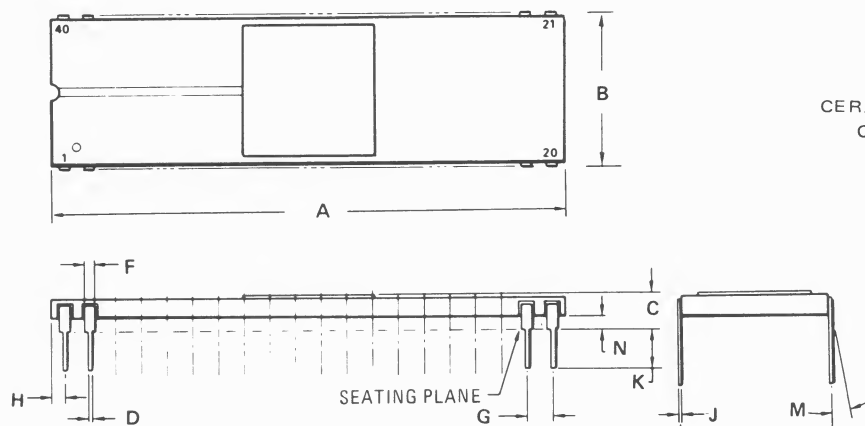
P SUFFIX
PLASTIC PACKAGE
CASE 711-03

NOTES:

1. POSITIONAL TOLERANCE OF LEADS (D), SHALL BE WITHIN 0.25 mm (0.010) AT MAXIMUM MATERIAL CONDITION, IN RELATION TO SEATING PLANE AND EACH OTHER.
2. DIMENSION L TO CENTER OF LEADS WHEN FORMED PARALLEL.
3. DIMENSION B DOES NOT INCLUDE MOLD FLASH.
4. 711-02 OBSOLETE, NEW STANDARD 711-03.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.69	52.45	2.035	2.065
B	13.72	14.22	0.540	0.560
C	3.94	5.08	0.155	0.200
D	0.36	0.56	0.014	0.022
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.38	0.008	0.015
K	2.92	3.43	0.115	0.135
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.02	0.020	0.040



L SUFFIX
CERAMIC PACKAGE
CASE 715-02

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
M	0°	10°	0°	10°
N	0.51	1.52	0.020	0.060

NOTE:

1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX. MAT'L CONDITION.





**MOTOROLA**

SEMICONDUCTORS

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

Advance Information

8-BIT MICROPROCESSING UNIT

The MC6809 is a revolutionary high performance 8-bit microprocessor which supports modern programming techniques such as position independence, reentrancy, and modular programming.

This third-generation addition to the MC6800 family has major architectural improvements which include additional registers, instructions and addressing modes.

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809 has the most complete set of addressing modes available on any microprocessor today.

The MC6809 has hardware and software features which make it an ideal processor for higher level language execution or standard controller applications.

MC6800 COMPATIBLE

- Hardware - Interfaces with All M6800 Peripherals
- Software - Upward Source Code Compatible Instruction Set and Addressing Modes

ARCHITECTURAL FEATURES

- Two 16-bit Index Registers
- Two 16-bit Indexable Stack Pointers
- Two 8-bit Accumulators can be concatenated to form one 16-bit Accumulator
- Direct Page Register allows Direct Addressing throughout memory map

HARDWARE FEATURES

- On chip oscillator (4 X fo XTAL)
- $\overline{\text{DMA/BREQ}}$ allows DMA operation or memory refresh
- Fast Interrupt Request Input stacks only Condition Code Register and Program Counter
- MRDY Input extends data access times for use with slow memory
- Interrupt Acknowledge output allows vectoring by devices
- SYNC Acknowledge output allows for synchronization to external event.
 - Single Bus-cycle RESET
 - Single 5-volt operation
 - NMI blocked after RESET until after first load of Stack Pointer
 - Early address valid allows use with slower memories

SOFTWARE FEATURES

- 10 Addressing Modes
 - 6800 Upward compatible Addressing Modes
 - Direct Addressing anywhere in memory map
 - Long Relative Branches
 - Program Counter Relative
 - Indirection
 - Expanded Index Addressing
 - 0,5,8,16-bit constant offsets
 - 8, 16-bit accumulator offsets
 - Auto-increment/decrement
- Improved Stack Manipulation
- 1464 Instructions with unique addressing modes
- 8 x 8 unsigned multiply
- 16-bit arithmetic
- Transfer/Exchange all registers
- Push/Pull **any** or **all** registers
- Load Effective Address

MC6809

(1.0 MHz)

MC68A09

(1.5 MHz)

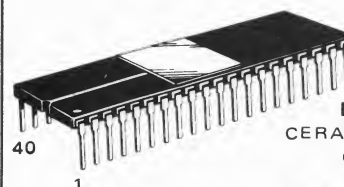
MC68B09

(2.0 MHz)

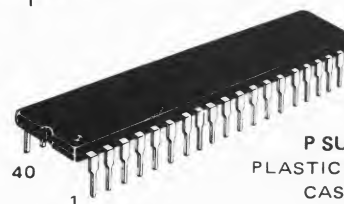
MOS

(N-CHANNEL, SILICON-GATE)

8-BIT MICROPROCESSING UNIT



L SUFFIX
CERAMIC PACKAGE
CASE 715



P SUFFIX
PLASTIC PACKAGE
CASE 711

PIN ASSIGNMENT

1	VSS	HALT	40
2	NMI	XTAL	39
3	IRQ	EXTAL	38
4	FIRO	RESET	37
5	BS	MRDY	36
6	BA	Q	35
7	VCC	E	34
8	A0	DMA/BREQ	33
9	A1	R/W	32
10	A2	D0	31
11	A3	D1	30
12	A4	D2	29
13	A5	D3	28
14	A6	D4	27
15	A7	D5	26
16	A8	D6	25
17	A9	D7	24
18	A10	A15	23
19	A11	A14	22
20	A12	A13	21

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V _{CC}	-0.3 to +7.0	Vdc
Input Voltage	V _{in}	-0.3 to +7.0	Vdc
Operating Temperature Range	T _A	0 to +70	°C
Storage Temperature Range	T _{stg}	-55 to +150	°C
Thermal Resistance	θ _{JA}	70	°C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 V ±5%, V_{SS} = 0, T_A = 0 to 70°C unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Logic, EXtal RESET	V _{IH}	V _{SS} + 2.0 V _{SS} + 4.0	— —	V _{DD} V _{DD}	Vdc
Input Low Voltage Logic, EXtal, RESET	V _{IL}	V _{SS} - 0.3	—	V _{SS} + 0.8	Vdc
Input Leakage Current (V _{in} = 0 to 5.25 V, V _{CC} = max)	I _{in}	—	1.0	2.5	μAdc
Output High Voltage (I _{Load} = -205 μAdc, V _{CC} = min) (I _{Load} = -145 μAdc, V _{CC} = min) (I _{Load} = -100 μAdc, V _{CC} = min)	V _{OH}	V _{SS} + 2.4 V _{SS} + 2.4 V _{SS} + 2.4	— — —	— — —	Vdc
Output Low Voltage (I _{Load} = 2.0 mAdc, V _{CC} = min)	V _{OL}	—	—	V _{SS} + 0.5	Vdc
Power Dissipation	P _D	—	—	1.0	W
Capacitance # (V _{in} = 0, T _A = 25°C, f = 1.0 MHz)	C _{in}	—	10	15	pF
	C _{out}	—	7 —	10 12	
Frequency of Operation (Crystal or External Input)	f f _{XTAL} f _{XTAL}	— — —	— — —	4 6 8	MHz
Three-State (Off State) Input Current (V _{in} = 0.4 to 2.4 V, V _{CC} = max)	I _{TSI}	— —	2.0 —	10 100	μAdc

READ/WRITE TIMING (Reference Figures 1 and 2)

Characteristic	Symbol	MC6809			MC68A09			MC68B09			Unit
		Min	Typ	Max	Min	Typ	Max	Min	Typ	Max	
Cycle Time	t _{CYC}	1000	—	—	667	—	—	500	—	—	ns
Total Up Time	t _{UT}	975	—	—	640	—	—	480	—	—	ns
Peripheral Read Access Time t _{ac} = (t _{AD} = t _{DSR})	t _{ACC}	695	—	—	440	—	—	320	—	—	ns
Data Setup Time (Read)	t _{DSR}	80	—	—	60	—	—	40	—	—	ns
Input Data Hold Time	t _{DHR}	10	—	—	10	—	—	10	—	—	ns
Output Data Hold Time	t _{DHW}	30	—	—	30	—	—	30	—	—	ns
Address Hold Time (Address, R/W)	t _{AH}	30	—	—	30	—	—	30	—	—	ns
Address Delay	t _{AD}	—	—	200	—	—	140	—	—	110	ns
Data Delay Time (Write)	t _{DDW}	—	—	225	—	—	180	—	—	145	ns
E _{low} to Q _{high} Time	t _{AVS}	—	—	250	—	—	165	—	—	125	ns
Address Valid to Q _{high}	t _{AQ}	25	—	—	25	—	—	15	—	—	ns
Processor Clock Low	t _{PWEL}	450	—	—	295	—	—	210	—	—	ns
Processor Clock High	t _{PWEH}	450	—	—	280	—	—	220	—	—	ns
MRDY Set Up Time	t _{PCSR}	60	—	—	60	—	—	60	—	—	ns
Interrupts Set Up Time	t _{PCS}	200	—	—	140	—	—	110	—	—	ns
HALT Set Up Time	t _{PCSH}	200	—	—	140	—	—	110	—	—	ns
RESET Set Up Time	t _{PCSR}	200	—	—	140	—	—	110	—	—	ns
DMA/BREQ Set Up Time	t _{PCSD}	125	—	—	125	—	—	125	—	—	ns
Crystal Osc Start Time	t _{rc}	100	—	—	100	—	—	100	—	—	ms
E _{rise} and Fall Time	t _{ER} , t _{EF}	5	—	25	5	—	25	5	—	20	ns
Processor Control Rise/Fall	t _{PCR} , t _{PLF}	—	—	100	—	—	100	—	—	100	ns
Q Rise and Fall Time	t _{QR} , t _{QF}	5	—	25	5	—	25	5	—	20	ns
Q Clock High	t _{PWQH}	450	—	—	280	—	—	220	—	—	ns



FIGURE 1 — READ DATA FROM MEMORY OR PERIPHERALS

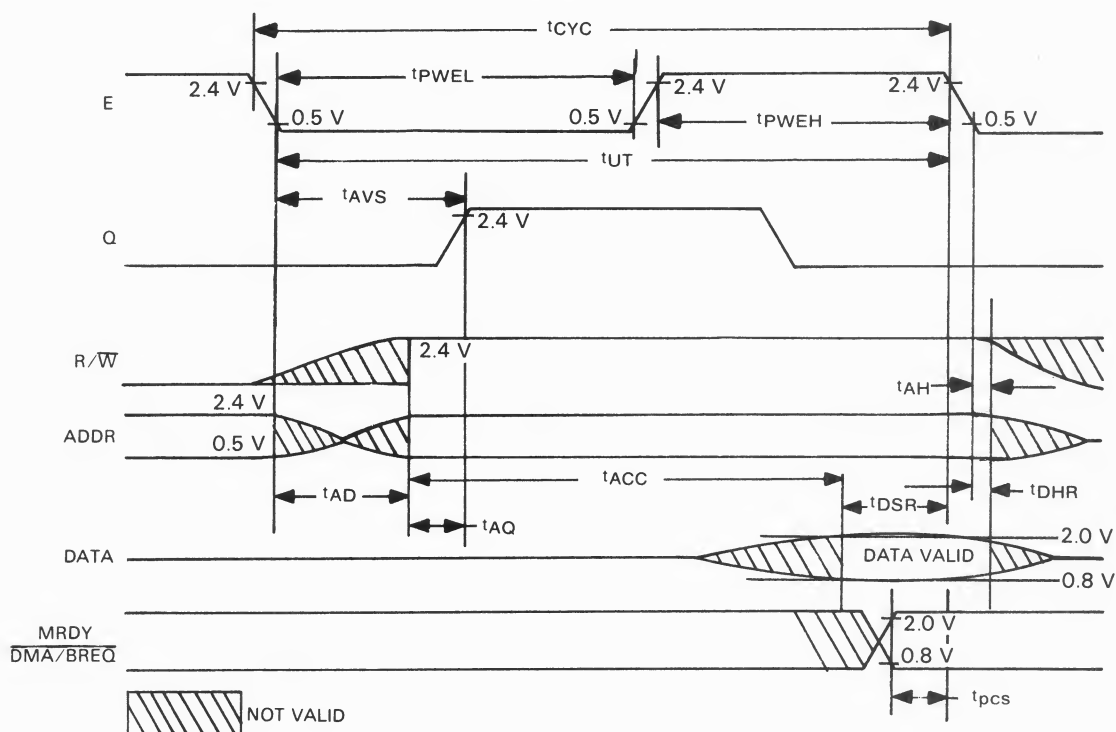


FIGURE 2 — WRITE DATA TO MEMORY OR PERIPHERALS

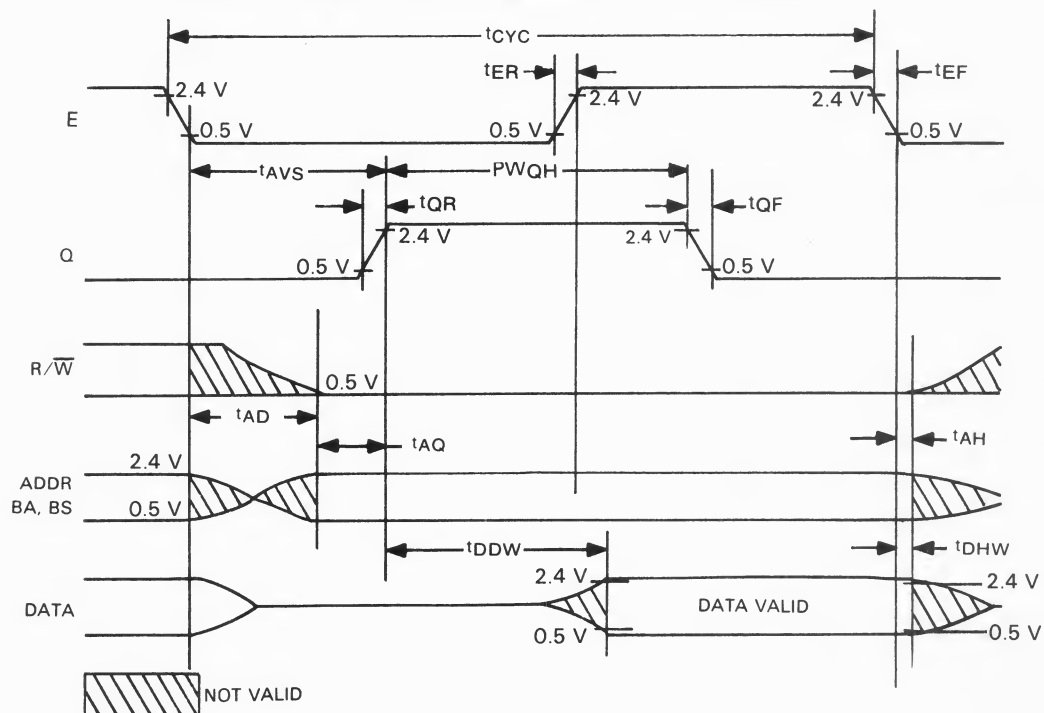


FIGURE 3 — MC6809 EXPANDED BLOCK DIAGRAM

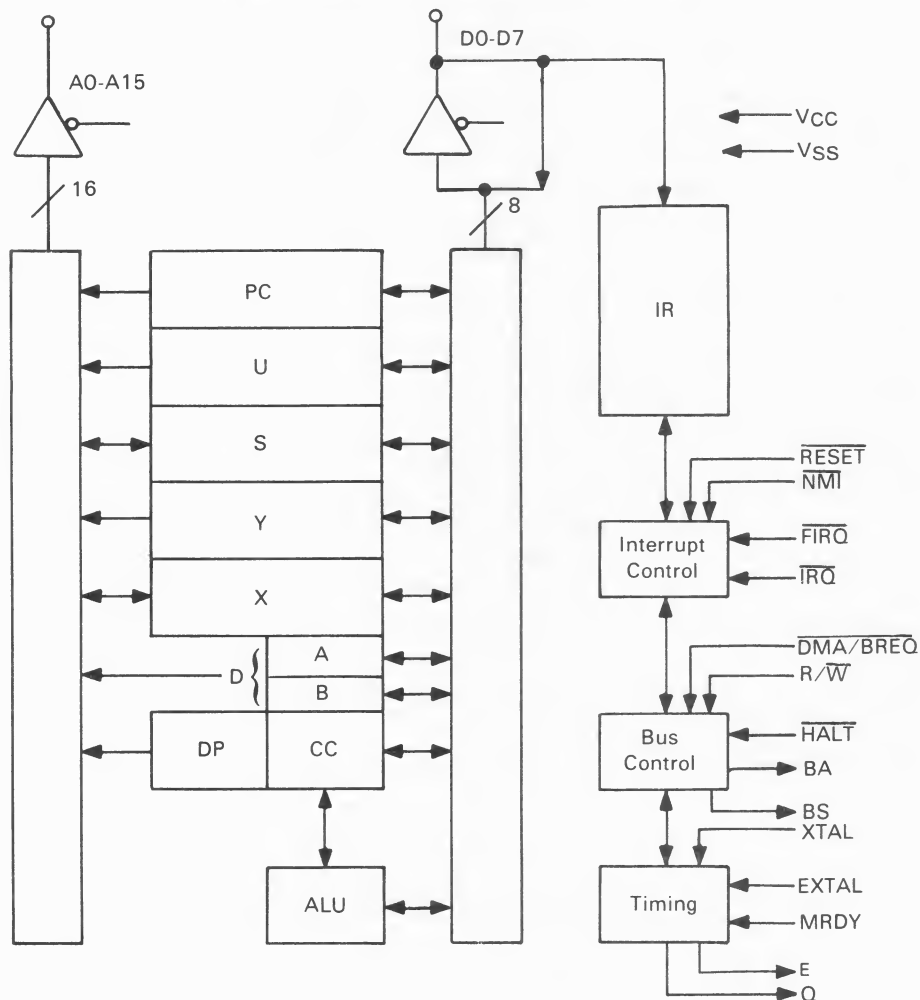
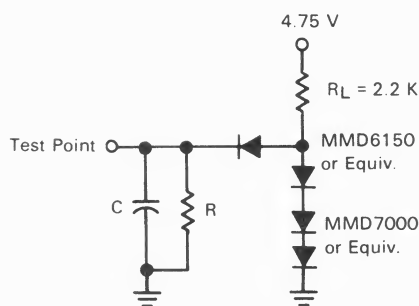


FIGURE 4 — BUS TIMING TEST LOAD



$C = 30 \text{ pF}$ for BA, BS
 130 pF for D0-D7, E, Q
 90 pF for A0-A15, R/W

$R = 11.7 \text{ k}\Omega$ for D0-D7
 $16.5 \text{ k}\Omega$ for A0-A15, E, Q
 $24 \text{ k}\Omega$ for BA, BS

PROGRAMMING MODEL

As shown in Figure 5, the MC6809 adds three registers to the set available in the MC6800. The added registers include a direct page register, the User Stack pointer and a second Index Register.

ACCUMULATORS (A, B, D)

The A and B registers are general purpose accumulators which are used for arithmetic calculations and manipulation of data.

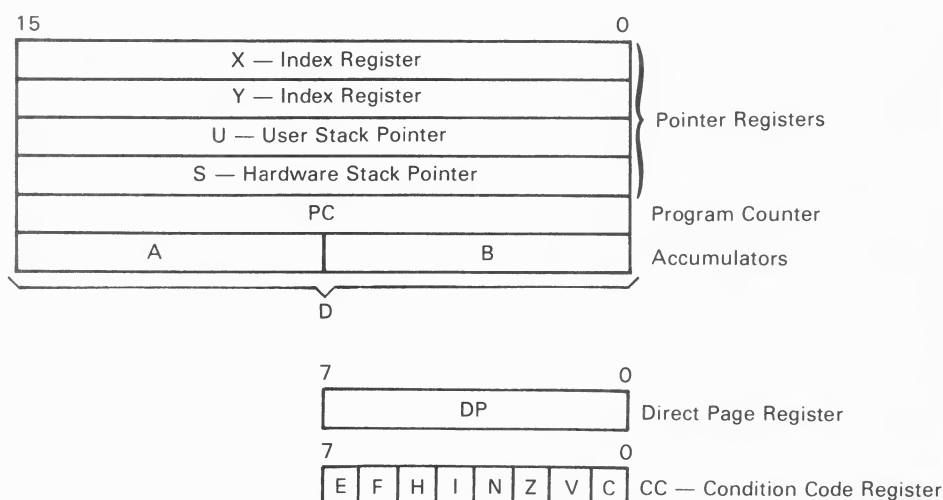
Certain instructions concatenate the A and B registers to form a single 16-bit accumulator. This is referred to as the D register, and is formed with the A register as the most significant byte.

DIRECT PAGE REGISTER (DP)

The Direct Page Register of the MC6809 serves to enhance the Direct Addressing Mode. The content of this register appears at the higher address outputs (A8-A15) during direct Addressing Instruction execution. This allows the direct mode to be used at any place in memory, under program control. To allow 6800 compatibility, all bits of this register are cleared during Processor Reset.



FIGURE 5 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT



INDEX REGISTERS (X,Y)

The Index Registers are used in indexed mode of addressing. The 16-bit address in this register takes part in the calculation of effective addresses. This address may be used to point to data directly or may be modified by an optional constant or register offset. During some indexed modes, the contents of the index register are incremented and decremented to point to the next item of tabular type data. All four pointer registers (X,Y,U,S) may be used as index registers.

STACK POINTERS (U,S)

The Hardware Stack Pointer (S) is used automatically by the processor during subroutine calls and interrupts. The stack pointers of the MC6809 point to the top of the stack, in contrast to the MC6800 stack pointer, which pointed to the next free location on the stack. The User Stack Pointer (U) is controlled exclusively by the programmer thus allowing arguments to be passed to and from subroutines with ease. Both Stack Pointers have the same indexed mode addressing capabilities as the X and Y registers, but also support **Push** and **Pull** instructions. This allows the MC6809 to be used efficiently as a stack processor, greatly enhancing its ability to support higher level languages and modular programming.

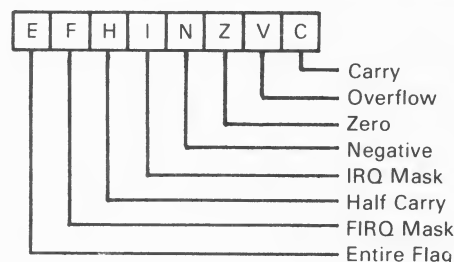
PROGRAM COUNTER

The Program Counter is used by the processor to point to the address of the next instruction to be executed by the processor. Relative Addressing is provided allowing the Program Counter to be used like an index register in some situations.

CONDITION CODE REGISTER

The condition code register defines the State of the Processor at any given time, see Figure 6.

FIGURE 6 — CONDITION CODE REGISTER FORMAT



CONDITION CODE REGISTER DESCRIPTION

BIT 0 (C)

Bit 0 is the Carry Flag, and is usually the carry from the binary ALU. C is also used to represent a 'borrow' from subtract like instructions (CMP, NEG, SUB, SBC). Here the carry flag is the complement of the carry from the binary ALU.

BIT 1 (V)

Bit 1 is the overflow flag, and is set to a one by an operation which causes a signed two's complement arithmetic overflow. This overflow is detected in an operation in which the carry from the MSB in the ALU does not match the carry from the MSB-1.

BIT 2 (Z)

Bit 2 is the zero flag, and is set to a one if the result of the previous operation was identically zero.



BIT 3 (N)

Bit 3 is the negative flag, which contains exactly the value of the MSB of the result of the preceding operation. Thus, a negative two's-complement result will leave N set to a one.

BIT 4 (I)

Bit 4 is the $\overline{\text{IRQ}}$ mask bit. The processor will not recognize interrupts from the IRQ line if this bit is set to a one. $\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, $\overline{\text{RESET}}$, and SWI all set I to a one; SWI2 AND SWI3 do not affect I.

BIT 5 (H)

Bit 5 is the half-carry bit, and is used to indicate a carry from bit 3 in the ALU as a result of an 8-bit addition only (ADC or ADD). This bit is used by the DAA instruction to perform a BCD decimal add adjust operation. The state of this flag is undefined in all subtract-like instructions.

BIT 6 (F)

Bit 6 is the $\overline{\text{FIRQ}}$ mask bit. The processor will not recognize interrupts from the FIRQ line if this bit is a one. $\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$, SWI, and $\overline{\text{RESET}}$ all set F to a one. $\overline{\text{IRQ}}$, SWI2 and SWI3 do not affect F.

BIT 7 (E)

Bit 7 is the entire flag, and when set to a one indicates that the complete machine state (all the registers) was stacked, as opposed to the subset state (PC and CC). The E bit of the stacked CC is used on a return from interrupt (RTI) to determine the extent of the unstacking. Therefore, the current E left in the Condition Code Register represents past action.

MC6809 MPU SIGNAL DESCRIPTION

POWER (V_{SS}, V_{CC})

Two pins are used to supply power to the part: V_{SS} is ground or 0 volts, while V_{CC} is +5.0 V \pm 5%.

ADDRESS BUS (A0-A15)

Sixteen pins are used to output address information from the MPU onto the Address Bus. When the processor does not require the bus for a data transfer, it will output address FFFF16, R/ $\overline{\text{W}}$ = 1, and BS = 0. Addresses are valid on the rising edge of Q (see Figure 1 and 2). All address bus drivers are made high-impedance when output Bus Available (BA) is high. Each pin will drive one Schottky TTL load and typically 90 pF.

DATA BUS (D0-D7)

These eight pins provide communication with the system bi-directional data bus. Each pin will drive one Schottky TTL load and typically 130 pF.

READ/WRITE (R/ $\overline{\text{W}}$)

This signal indicates the direction of data transfer on the data bus. A low indicates that the MPU is writing data onto the data bus. R/ $\overline{\text{W}}$ is made high impedance when BA is high. R/ $\overline{\text{W}}$ is valid on the rising edge of Q, refer to Figure 1 and 2.

 $\overline{\text{RESET}}$

A low level on this Schmitt-trigger input for greater than one bus cycle will reset the MPU as shown Figure 7. The Reset vectors are fetched from locations FFFE16 and FFFF16 (Table 1) when Interrupt Acknowledge is true, (BA \wedge BS = 1). During initial power-on, the Reset line should be held low until the clock oscillator is fully operational; see Figure 8.

Because the MC6809 Reset pin has a Schmitt-trigger input with a threshold voltage higher than that of standard peripherals, a simple R/C network may be used to reset the entire system. This higher threshold voltage insures that all peripherals are out of the reset state before the Processor.

 $\overline{\text{HALT}}$

A low level on this input pin will cause the MPU to stop running at the end of the present instruction and remain halted indefinitely without loss of data. When Halted, the BA output is driven high indicating the buses are high-impedance. BS is also high which indicates the processor is in the Halt or Bus Grant state. While halted, the MPU will not respond to external real-time requests ($\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$) although $\overline{\text{DMA/BREQ}}$ will always be accepted, and $\overline{\text{NMI}}$ or $\overline{\text{RESET}}$ will be latched for later response. During the Halt state Q and E continue to run normally. If the MPU is not running ($\overline{\text{RESET}}$, $\overline{\text{DMA/BREQ}}$), a halted state (BA and BS = 1) can be achieved by pulling $\overline{\text{HALT}}$ low while $\overline{\text{RESET}}$ is still low. If $\overline{\text{DMA/BREQ}}$ and $\overline{\text{HALT}}$ are both pulled low, the processor will reach the last cycle of the instruction (by reverse cycle stealing) where the machine will then become halted. See Figure 9.

BUS AVAILABLE, BUS STATUS (BA, BS)

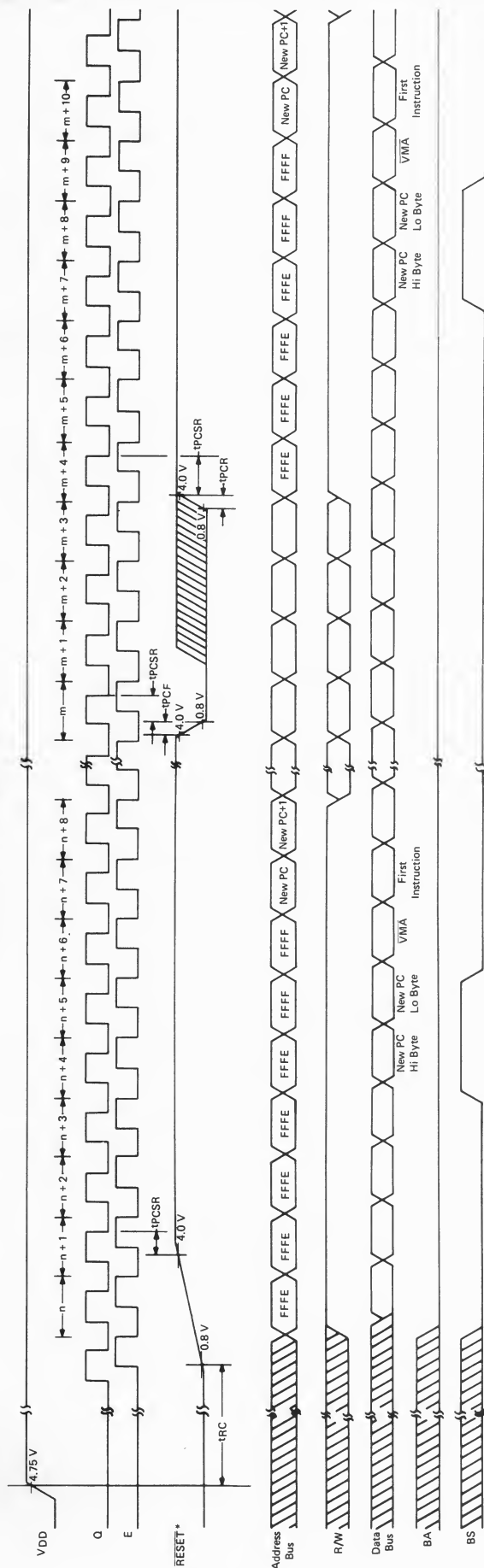
The Bus Available output is an indication of an internal control signal which makes the MOS buses of the MPU high-impedance. This signal does not imply that the bus will be available for more than one cycle. When BA goes low, an additional dead cycle will elapse before the MPU acquires the bus.

The Bus Status output signal, when decoded with BA, represents the MPU state (valid with leading edge of Q):

MPU State		
BA	BS	
0	0	Normal (Running)
0	1	Interrupt Acknowledge
1	0	SYNC Acknowledge
1	1	HALT or Bus Grant

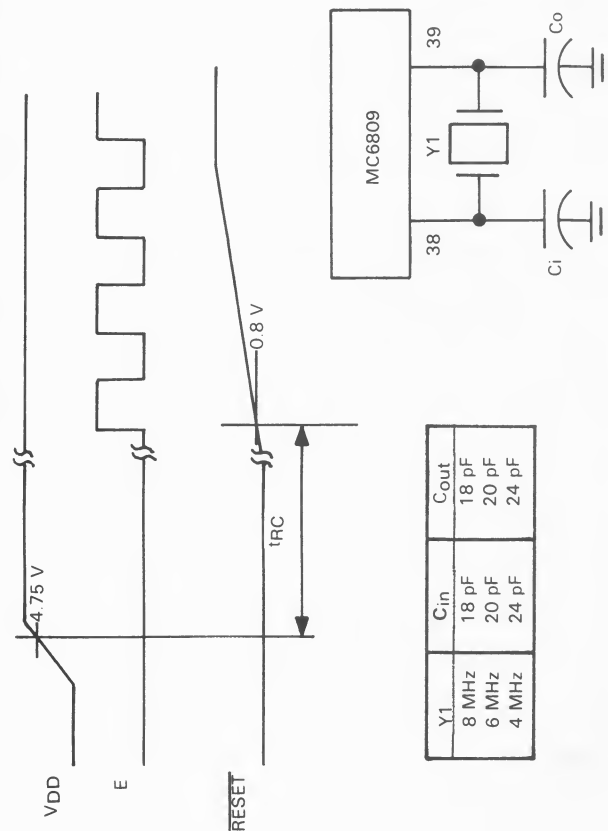


FIGURE 7 — RESET TIMING



*Note: Parts with date codes prefixed by 7F will come out of Reset one cycle sooner than shown.

FIGURE 8 — CRYSTAL CONNECTIONS AND OSCILLATOR START UP



Y1	C _{in}	C _{out}
8 MHz	18 pF	18 pF
6 MHz	20 pF	20 pF
4 MHz	24 pF	24 pF

6809 Crystal Parameters*

	3.58 MHz	4.00 MHz	6.0 MHz	8.0 MHz
RS	60 Ω	50 Ω	30-50 Ω	20-40 Ω
C ₀	3.5 pF	6.5 pF	4-6 pF	4-6 pF
C ₁	.015 pF	.025 pF	.01-.02 pF	.01-.02 pF
C _{in} , C _{out}	25 pF	25 pF	25 pF	25 pF
Q	40 K	30 K	≈20 K	≈20 K

All Parameters Are ±10%

*Note: These are representative AT-cut crystal parameters only. Crystals of other types of cut that work may also be used.

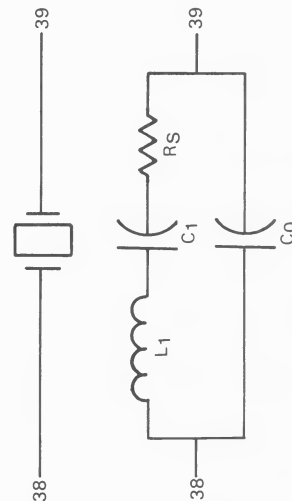
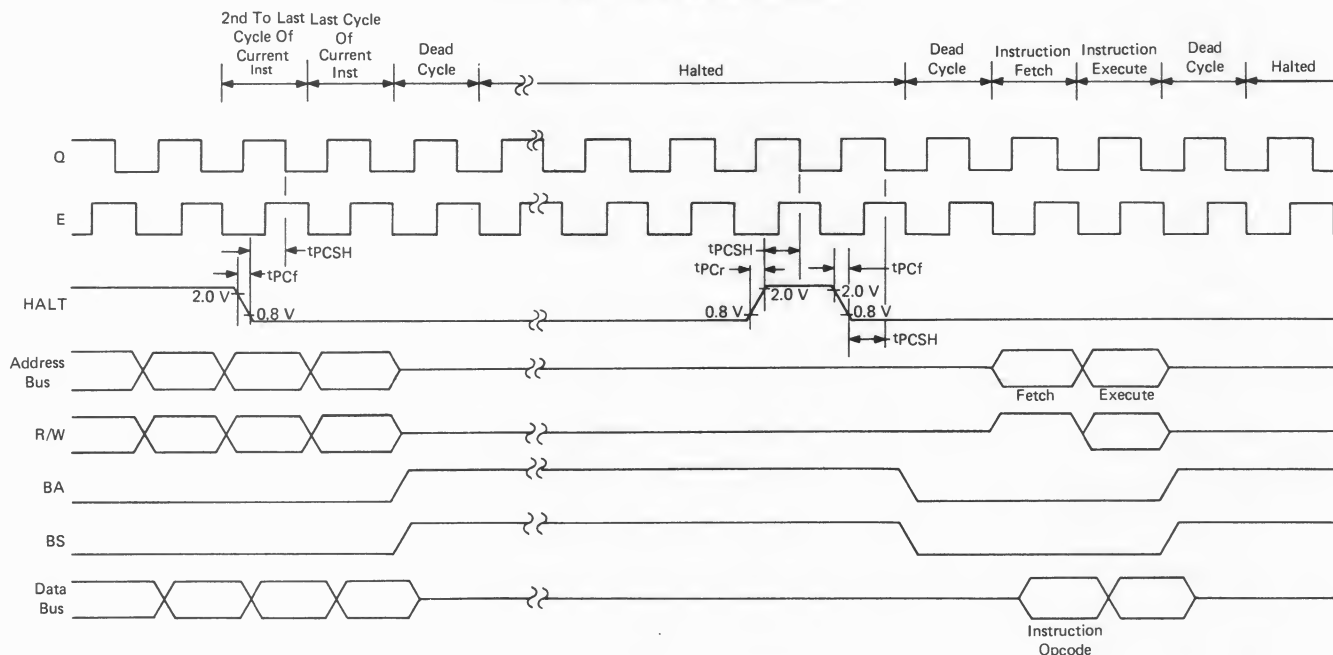


FIGURE 9 — $\overline{\text{HALT}}$ AND SINGLE INSTRUCTION EXECUTION FOR SYSTEM DEBUG

Interrupt Acknowledge is indicated during both cycles of a hardware-vector-fetch ($\overline{\text{RESET}}$, $\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, SWI , SWI2 , SWI3). This signal, plus decoding of the lower 4 address lines can provide the user with an indication of which interrupt level is being serviced and allow vectoring by device, (see Table 1).

Sync Acknowledge is indicated while the MPU is waiting for external synchronization on an interrupt line.

Halt/Bus Grant is true when the MC6809 is in a Halt or Bus Grant condition.

TABLE 1: MEMORY MAP FOR INTERRUPT VECTORS

Memory Map For Vector Location		Interrupt Vector Description
MS	LS	
FFFE	FFFF	$\overline{\text{RESET}}$
FFFC	FFFD	$\overline{\text{NMI}}$
FFFA	FFFB	SWI
FFF8	FFF9	$\overline{\text{IRQ}}$
FFF6	FFF7	$\overline{\text{FIRQ}}$
FFF4	FFF5	SWI2
FFF2	FFF3	SWI3
FFF0	FFF1	Reserved

***NOTE:** $\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$ and $\overline{\text{IRQ}}$ requests are latched by the falling edge of every Q except during cycle stealing operations (e.g., DMA) where only $\overline{\text{NMI}}$ is latched. From this point, a delay of at least one bus cycle will occur before the interrupt is serviced by the MPU.

NON MASKABLE INTERRUPT ($\overline{\text{NMI}}$)

A negative edge on this input requests that a non-maskable interrupt sequence be generated. A non-maskable interrupt cannot be inhibited by the program, and also has a higher priority than $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$ or software interrupts. During recognition of an $\overline{\text{NMI}}$, the entire machine state is saved on the hardware stack. After reset, an $\overline{\text{NMI}}$ will not be recognized until the first program load of the Hardware Stack Pointer (S). The pulse width of $\overline{\text{NMI}}$ low must be at least one E cycle. If the $\overline{\text{NMI}}$ input does not meet the minimum set up with respect to Q, the interrupt will not be recognized until the next cycle. See Figure 10.

FAST-INTERRUPT REQUEST ($\overline{\text{FIRQ}}$)

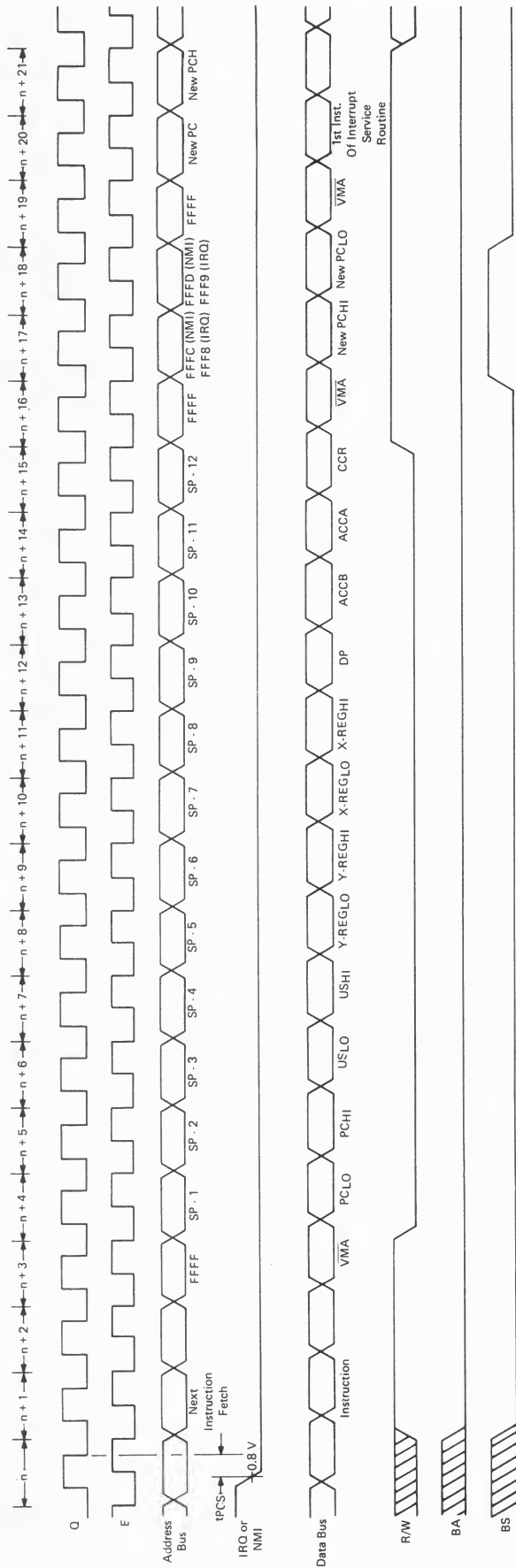
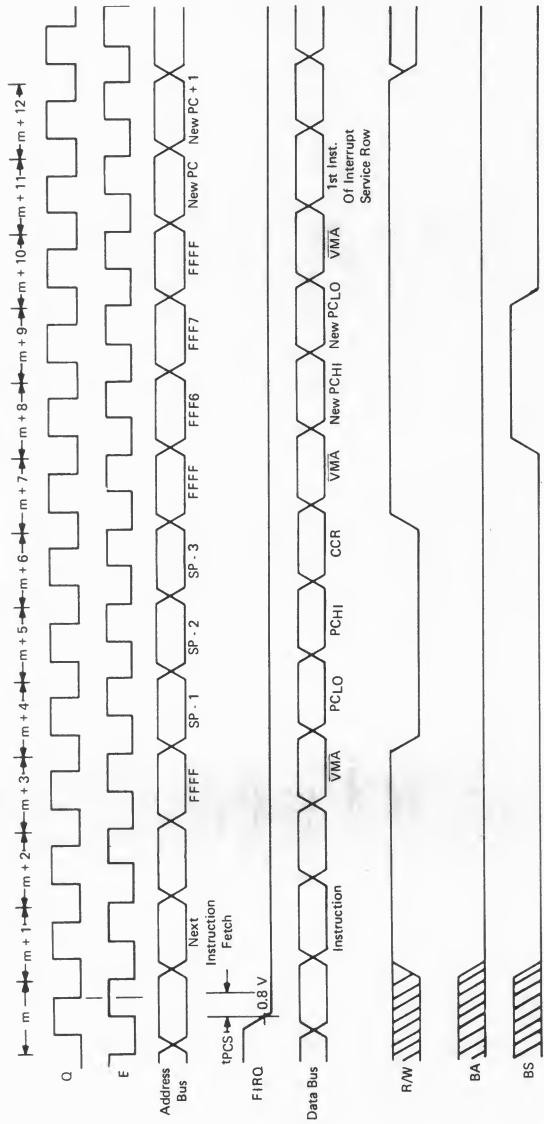
A low level on this input pin will initiate a fast interrupt sequence, provided its mask bit (F) in the CC is clear. This sequence has priority over the standard Interrupt Request ($\overline{\text{IRQ}}$), and is fast in the sense that it stacks only the contents of the condition code register and the program counter. The interrupt service routine should clear the source of the interrupt before doing an RTI. See Figure 11.

INTERRUPT REQUEST ($\overline{\text{IRQ}}$)

A low level input on this pin will initiate an Interrupt Request sequence provided the mask bit (I) in the CC is clear. Since $\overline{\text{IRQ}}$ stacks the entire machine state it provides a slower response to interrupts than $\overline{\text{FIRQ}}$. $\overline{\text{IRQ}}$ also has a lower priority than $\overline{\text{FIRQ}}$. Again, the interrupt service routine should clear the source of the interrupt before doing an RTI. See Figure 10.



FIGURE 10 — $\overline{\text{IRQ}}$ AND $\overline{\text{NMI}}$ INTERRUPT TIMING

FIGURE 11 – $\overline{\text{FIRQ}}$ INTERRUPT TIMING

XTAL, EXTAL

These input pins are used to connect the on-chip oscillator to an external parallel-resonant crystal. Alternately, the pin EXTAL may be used as a TTL level input for external timing by grounding XTAL. The crystal or external frequency is 4 times the bus frequency, see Figure 8. Proper RF layout techniques should be observed in the layout of printed circuit boards.

E, Q

E is similar to the MC6800 bus timing signal ϕ_2 ; Q is a quadrature clock signal which leads E. Q has no parallel on the MC6800. Addresses from the MPU will be valid with the leading edge of Q. Data is latched on the falling edge of E. Timing for E and Q is shown in Figure 12.

MRDY

This input control signal allows stretching of E to extend data-access time. When MRDY is high, E will be in normal operation. When MRDY is low, E may be stretched integral multiples of quarter ($\frac{1}{4}$) bus cycles, thus allowing interface to slow memories as shown in Figure 13. A maximum stretch is 10 microseconds. During non-valid memory accesses ($\overline{\text{VMA}}$ cycles), MRDY has no effect on stretching E. This inhibits slowing the processor speed during "don't care" bus accesses.

DMA/BREQ

The $\overline{\text{DMA/BREQ}}$ input provides a method of suspending execution and acquiring the MPU bus for another use as shown in Figure 14. Typical uses include DMA and dynamic memory refresh.

Transition of $\overline{\text{DMA/BREQ}}$ should occur during Q. A low level on this pin will stop instruction execution at the end of the current cycle. The MPU will acknowledge $\overline{\text{DMA/BREQ}}$ by setting BA and BS to a one. The requesting device will now have up to 15 bus cycles before the MPU retrieves the bus for self-refresh. Self-refresh requires one bus cycle with a leading and trailing dead cycle, see Figure 15.

Typically, the DMA controller will request to use the bus by asserting the $\overline{\text{DMA/BREQ}}$ pin low on the leading edge of E. When the MPU replies with BA = BS = 1, that cycle will be a dead cycle used to transfer control to the DMA controller.

False memory accesses should be prevented during any dead cycles. When BA is cleared (either as a result of $\overline{\text{DMA/BREQ}} = \text{HIGH}$ or MPU self-refresh), the DMA device should be taken off the bus.

Another dead cycle will elapse before the MPU is allowed a memory access to transfer control without contention.

MPU OPERATION

During normal operation, the MPU fetches an instruction from memory and then executes the requested function. This sequence begins at RESET and is repeated indefinitely unless altered by a special instruction or hardware occurrence. Software instructions that alter normal MPU operation are: SWI, SWI2, SWI3, CWA, RTI and SYNC. An interrupt, HALT or $\overline{\text{DMA/BREQ}}$ can also alter the normal execution of instructions. Figure 16 illustrates the flow chart for the MC6809. The left-half of the flow chart represents normal operation; the right-half represents the flow when an interrupt or special instruction occurs.

FIGURE 12 — E/Q RELATIONSHIP

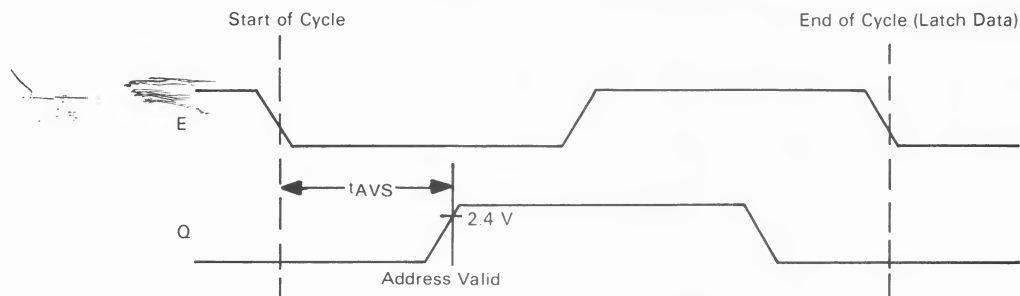


FIGURE 13 — MRDY TIMING

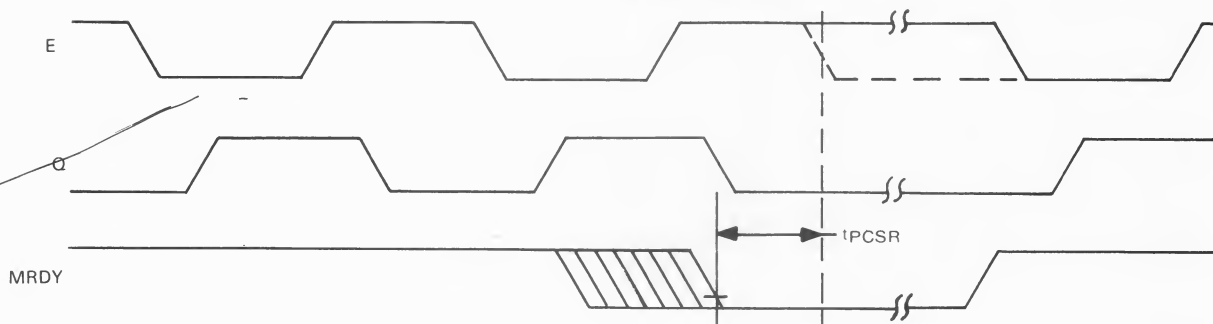
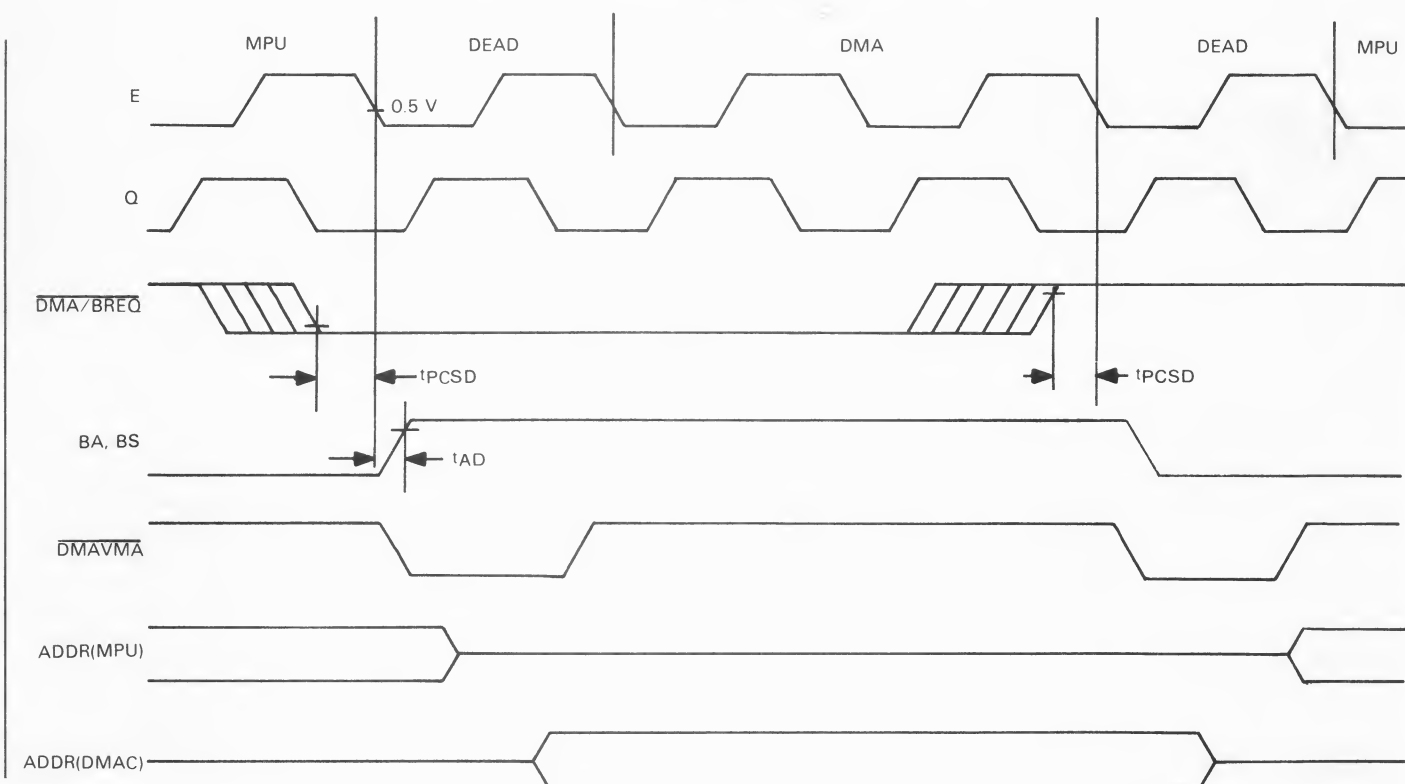


FIGURE 14 — TYPICAL DMA TIMING (<14 CYCLES)

**NOTE:**

DMAVMA is a signal which is developed externally, but is a system requirement for DMA.

FIGURE 15 — AUTO-REFRESH DMA TIMING (>14 CYCLES)

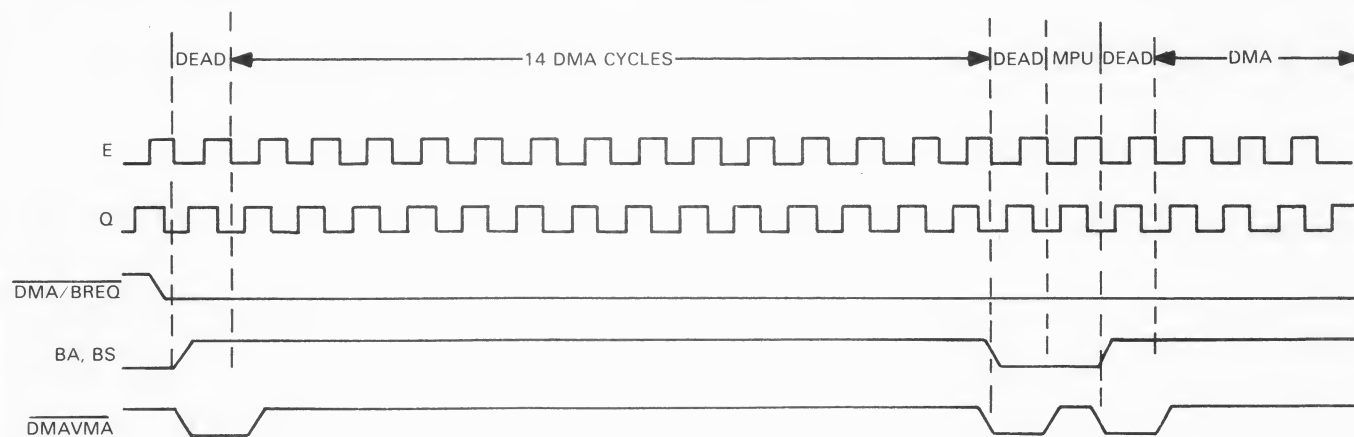
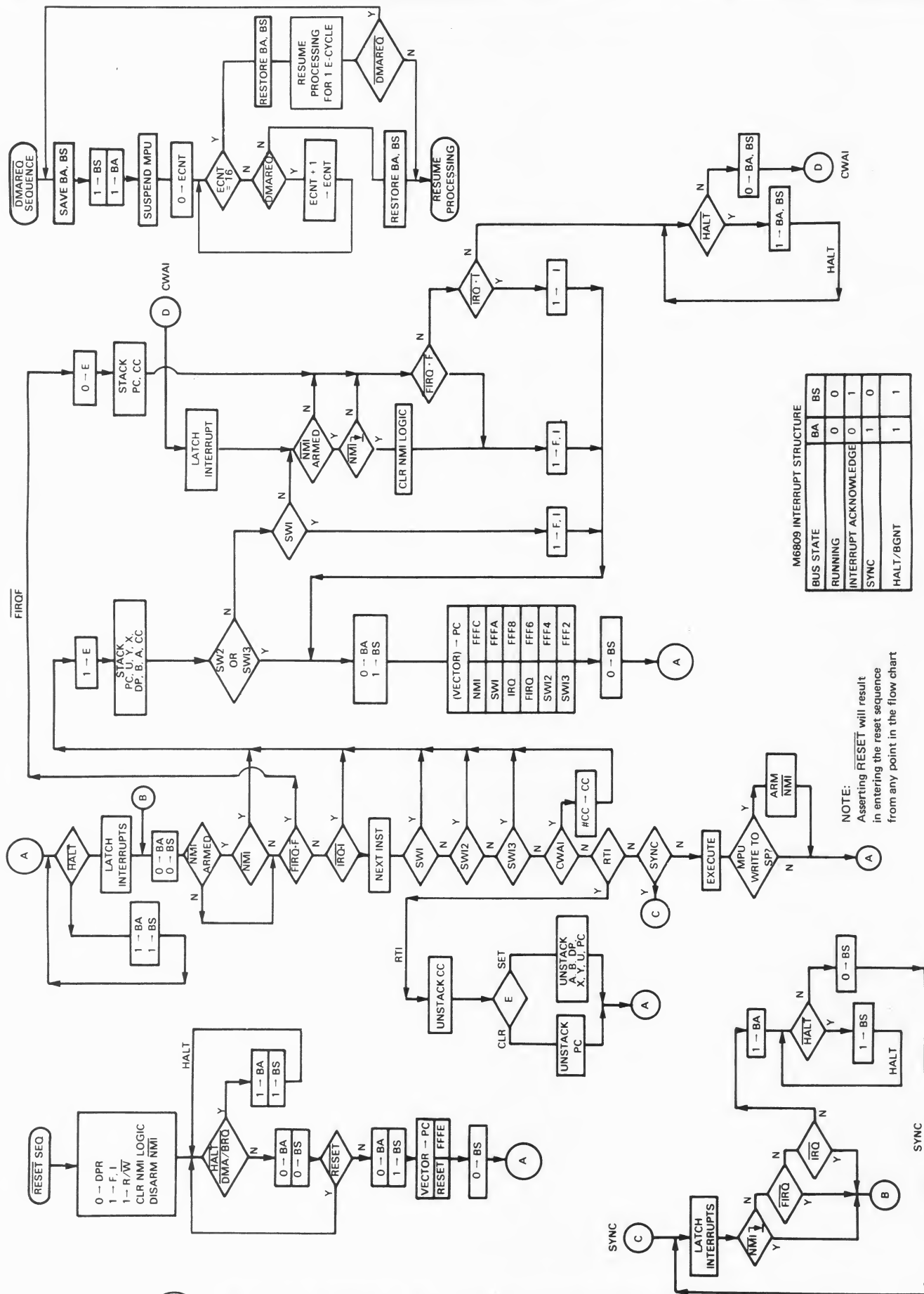


FIGURE 16 — MPU FLOWCHART



M6809 INTERRUPT STRUCTURE		
BUS STATE	BA	BS
RUNNING	0	0
INTERRUPT ACKNOWLEDGE	0	1
SYNC	1	0
HALT/BGNT	1	1

NOTE: Asserting RESET will result in entering the reset sequence from any point in the flow chart.

ADDRESSING MODES

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809 has the most complete set of addressing modes available on any microcomputer today. For example, the MC6809 has 59 basic instructions, however it recognizes 1464 different variations of instructions and addressing modes. The new addressing modes support modern programming techniques. The following addressing modes are available on the MC6809:

- Inherent (Includes Accumulator)
- Immediate
- Extended
 - Extended Indirect
- Direct
- Register
- Indexed
 - Zero-Offset
 - Constant Offset
 - Accumulator Offset
 - Auto Increment/Decrement
 - Indexed Indirect
- Relative
 - Short/Long Relative Branching
 - Program Counter Relative Addressing

INHERENT (INCLUDES ACCUMULATOR)

In this addressing mode, the opcode of the instruction contains all the address information necessary. Examples of Inherent Addressing are: ABX, DAA, SWI, ASRA, and CLRB.

IMMEDIATE ADDRESSING

In Immediate Addressing, the effective address of the data is the location immediately following the opcode; the data to be used in the instruction immediately follows the opcode of the instruction. The MC6809 uses both 8 and 16-bit immediate values depending on the size of argument specified by the opcode. Examples of instructions with Immediate Addressing are:

```
LDA #$20
LDX #$F000
LDY #CAT
```

Note: #. signifies Immediate addressing, \$ signifies hexadecimal value

EXTENDED ADDRESSING

In Extended Addressing the contents of the two bytes immediately following the opcode fully specify the 16-bit effective address used by the instruction. Note that the address generated by an extended instruction defines an absolute address and is not position independent. Examples of Extended Addressing include:

```
LDA CAT
STX MOUSE
LDD $2000
```

EXTENDED INDIRECT

As a special case of indexed addressing (discussed below), one level of indirection may be added to Extended Addressing. In Extended Indirect, the two bytes following the postbyte of an Indexed instruction contains the address of the address of the data.

```
LDA [CAT]
LDX [$FFFE]
STU [DOG]
```

DIRECT ADDRESSING

Direct addressing is similar to extended addressing except that only one byte of address follows the opcode. This byte specifies the lower 8 bits of the address to be used. The upper 8 bits of the address are supplied by the direct page register. Since only one byte of address is required in direct addressing, this mode requires less memory and executes faster than extended addressing. Of course, only 256 locations (one page) can be accessed without redefining the contents of the DP register. Since the DP register is set to \$00 on Reset, direct addressing on the MC6809 is compatible with direct addressing on the M6800. Indirection is not allowed in direct addressing. Some examples of direct addressing are:

```
LDA    $30
SETDP  $10 (Assembler directive)
LDB    $1030
LDD    < CAT
```

Note: < is an assembler directive which forces direct addressing.

REGISTER ADDRESSING

Some opcodes are followed by a byte that defines a register or set of registers to be used by the instruction, this is called a POSTBYTE. Some examples of register addressing are:

TFR	X,Y	Transfers X into Y
EXG	A,B	Exchanges A with B
PSHS	A,B,X,Y	Push onto S Y,X,B, then A
PULU	X,Y,D	Pull from U D,X, then Y

INDEXED ADDRESSING

In all indexed addressing one of the pointer registers (X, Y, U, S, and sometimes PC) is used in a calculation of the effective address of the operand to be used by the instruction. Five basic types of indexing are available and are discussed below. The postbyte of an indexed instruction specifies the basic type and variation of the addressing mode as well as the pointer register to be used. Figure 17 lists the legal formats for the postbyte. Table 2 gives the assembler form and the number of cycles and bytes added to the basic values for indexed addressing for each variation.



FIGURE 17 — INDEXED ADDRESSING POSTBYTE REGISTER BIT ASSIGNMENTS

Post-Byte Register Bit								Indexed Addressing Mode
7	6	5	4	3	2	1	0	
0	R	R	X	X	X	X	X	EA = ,R ± 4 Bit Offset
1	R	R	0	0	0	0	0	,R+
1	R	R	1	0	0	0	1	,R++
1	R	R	0	0	0	1	0	,-R
1	R	R	1	0	0	1	1	,--R
1	R	R	1	0	1	0	0	EA = ,R ± 0 Offset
1	R	R	1	0	1	0	1	EA = ,R ± ACCB Offset
1	R	R	1	0	1	1	0	EA = ,R ± ACCA Offset
1	R	R	1	1	0	0	0	EA = ,R ± 7 Bit Offset
1	R	R	1	1	0	0	1	EA = ,R ± 15 Bit Offset
1	R	R	1	1	0	1	1	EA = ,R ± D Offset
1	X	X	1	1	1	0	0	EA = ,PC ± 7 Bit Offset
1	X	X	1	1	1	0	1	EA = ,PC ± 15 Bit Offset
1	R	R	1	1	1	1	1	EA = ,Address

Addressing Mode Field

Indirect Field
Sign bit when B7 = 0

Register Field

00:R = X

01:R = Y

10:R = U

11:R = S

X = Don't Care

Zero-Offset Indexed — In this mode, the selected pointer register contains the effective address of the data to be used by the instruction. This is the fastest indexing mode.

Examples are:

LDD 0,X

LDA 0,S

Constant Offset Indexed — In this mode a two's-complement offset and the contents of one of the pointer registers are added to form the effective address of the operand. The pointer register's initial content is unchanged by the addition.

Three sizes of offsets are available:

± 4-bit (-16 to +15)

± 7-bit (-128 to +127)

± 15-bit (-32768 to +32767)

The two's complement 5-bit offset is included in the postbyte and therefore is most efficient in use of bytes and cycles. The two's complement 8-bit offset is contained in a single byte following the postbyte. The two's complement 16-bit offset is in the two bytes following the postbyte. In most cases the programmer need not be concerned with the size of this offset since the assembler will select the optional size automatically.

Examples of constant-offset indexing are:

LDA 23,X

LDX -2,S

LDY 300,X

LDU CAT,Y

TABLE 2 — INDEXED ADDRESSING MODES

Type	Forms	Non Indirect				Indirect			
		Assembler Form	Postbyte OP Code	x ~	+ #	Assembler Form	Postbyte OP Code	+ ~	+ #
Constant Offset From R (Signed Offsets)	No Offset	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5 Bit Offset	n, R	0RRnnnnn	1	0	defaults to 8-bit			
	8 Bit Offset	n, R	1RR01000	1	1	[n, R]	1RR11000	4	1
	16 Bit Offset	n, R	1RR01001	4	2	[n, R]	1RR11001	7	2
Accumulator Offset From R (Signed Offsets)	A — Register Offset	A, R	1RR00110	1	0	[A, R]	1RR10110	4	0
	B — Register Offset	B, R	1RR00101	1	0	[B, R]	1RR10101	4	0
	D — Register Offset	D, R	1RR01011	4	0	[D, R]	1RR11011	7	0
Auto Increment/Decrement R	Increment By 1	,R+	1RR00000	2	0	not allowed			
	Increment By 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	Decrement By 1	,-R	1RR00010	2	0	not allowed			
	Decrement By 2	,--R	1RR00011	3	0	[,-R]	1RR10011	6	0
Constant Offset From PC	8 Bit Offset	n, PCR	1XX01100	1	1	[n, PCR]	1XX11100	4	1
	16 Bit Offset	n, PCR	1XX01101	5	2	[n, PCR]	1XX11101	8	2
Extended Indirect	16 Bit Address	—	—	—	—	[n]	10011111	5	2

R = X, Y, U or S X = 00 Y = 01

X = Don't Care U = 10 S = 11

+ and + Indicate the number of additional cycles and bytes for the particular variation.
~ #

Accumulator-Offset Indexed — This mode is similar to constant offset indexed except that the two's-complement value in one of the accumulators (A, B or D) and the content of one of the pointer registers are added to form the effective address of the operand. The contents of both the accumulator and the pointer register are unchanged by the addition. The postbyte specifies which accumulator to use as an offset and no additional bytes are required. The advantage of an accumulator offset is that the value of the offset can be calculated by a program at run-time.

Some examples are:

```
LDA B,Y
LDX D,Y
LEAX B,X
```

Auto Increment/Decrement Indexed — In the auto increment addressing mode, the pointer register contains the address of the operand. Then, after the pointer register is used it is incremented by one or two. This addressing mode is useful in stepping through tables, moving data, or for the creation of software stacks. In auto decrement, the pointer register is decremented prior to use as the address of the data. The use of auto decrement is similar to that of auto increment but the tables, etc. are scanned from the high to low addresses. The size of the increment/decrement can be either one or two to allow for tables of either 8 or 16-bit data to be accessed and is selectable by the programmer. The pre-decrement, post-increment nature of these modes allow them to be used to create additional software stacks that behave identically to the U and S stacks.

Some examples of the auto increment/decrement addressing modes are:

```
LDA ,X+
STD ,Y++
LDB ,-Y
LDX ,--S
```

INDEXED INDIRECT

All of the indexing modes with the exception of auto increment/decrement by one, or a ± 4 -bit offset may have an additional level of indirection specified. In Indirect addressing, the effective address is contained at the location specified by the content of the Index register plus any offset. In the example below, the A accumulator is loaded indirectly using an effective address calculated from the Index register and an offset.

```
Before Execution
A = XX (don't care)
X = $F000
$0100 LDA [10, X] EA is now $F010

$F010 $F1
$F011 $50 F150 is now the
new EA

$F150 $AA
```

After Execution
A = \$AA Actual Data Loaded

All modes of indexed indirect are included except those which are meaningless (e.g. auto increment/decrement by 1 indirect). Some examples of indexed indirect are:

```
LDA [X]
LDD [10,S]
LDA [B,Y]
LDD [X++]
```

RELATIVE ADDRESSING

The byte(s) following the branch opcode is (are) treated as a signed offset which is added to the program counter. If the branch condition is true then the calculated address (PC + signed offset) is loaded into the program counter. Program execution continues at the new location as indicated by the PC; Short (1 byte offset) and long (2 bytes offset) relative addressing modes are available. All of memory can be reached in long relative addressing as an effective address is interpreted modulo 2^{16} . Some examples of relative addressing are:

```
BEQ CAT (short)
BGT DOG (short)
CAT LBEQ RAT (long)
DOG LBGT RABBIT (long)
.
.
.
RAT NOP
RABBIT NOP
```

PROGRAM COUNTER RELATIVE

The PC can be used as the pointer register with 8 or 16-bit signed offsets. As in relative addressing the offset is added to the current PC to create the effective address. The effective address is then used as the address of the operand or data. Program Counter Relative Addressing is used for writing position independent programs. Tables related to a particular routine will maintain the same relationship after the routine is moved, if referenced relative to the Program Counter. Examples are:

```
LDA CAT, PCR
LEAX TABLE, PCR
```

Since program counter relative is a type of indexing, an additional level of indirection is available.

```
LDA [CAT, PCR]
LDU [DOG, PCR]
```



MC6809 INSTRUCTION SET

The instruction set of the MC6809 is similar to that of the MC6800 and is upward compatible at the source code level. The number of opcodes has been reduced from 72 to 59, but because of the expanded architecture and additional addressing modes, the number of available opcodes (with different addressing modes) has risen from 197 to 1464.

Some of the new instructions and addressing modes are described in detail below:

PSHU/PSHS

The push instructions have the capability of pushing onto either the hardware stack (S) or user stack (U) any or all of the MPU registers with a single instruction.

PULU/PULS

The pull instructions have the same capability of the push instruction, in reverse order. The byte immediate following the push or pull opcode determines which register or registers are to be pushed or pulled. The actual PUSH/PULL sequence is fixed; each bit defines a unique register to push or pull as shown in Figure 16.

TFR/EXG

Within the MC6809, any register may be transferred to or exchanged with another of like-size, i.e. 8-bit to 8-bit or 16-bit to 16-bit. Bits 4-7 of postbyte define the source

register, while bits 0-3 represent the destination register. These are denoted as follows:

0000 — D	0101 — PC
0001 — X	1000 — A
0010 — Y	1001 — B
0011 — U	1010 — CC
0100 — S	1011 — DP

Note: All other combinations are undefined and INVALID.

Load Effective Address

The LEA works by calculating the effective address used in an indexed instruction and stores that address value, rather than the data at that address, in a pointer register. This makes all the features of the internal addressing hardware available to the programmer. Some of the implications of this instruction are illustrated in the following table of examples:

The LEA instruction also allows the user to access data in a position independent manner. For example:

```
LEAX  MSG1, PCR
LBSR  PDATA (Print message routine)
```

```
MSG1 FCC  'MESSAGE'
```

FIGURE 16 — PUSH/PULL POSTBYTE

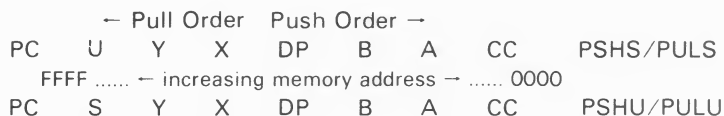


TABLE 3 — LEA EXAMPLES

Instruction		Operation		Comment
LEAX	10, X	X + 10	→ X	Adds 5-bit constant 10 to X
LEAX	500, X	X + 500	→ X	Adds 16-bit constant 500 to X
LEAY	A, Y	Y + A	→ Y	Adds 8-bit accumulator to Y
LEAY	D, Y	Y + D	→ Y	Adds 16-bit D accumulator to Y
LEAU	-10, U	U - 10	→ U	Subtracts 10 from U
LEAS	-10, S	S - 10	→ S	Used to reserve area on stack
LEAS	10, S	S + 10	→ S	Used to 'clean up' stack
LEAX	5, S	S + 5	→ X	Transfers as well as adds



This sample program prints "message". By writing MSG1,PCR, the assembler computes the distance between the present address and MSG1. This result is placed as a constant into the LEAX instruction which will be indexed from the PC value at the time of execution. No matter where the code is located, when it is executed, the computed offset from the PC will put the absolute address of MSG1 into the X pointer register. This code is totally position independent.

MUL

Multiplies the unsigned binary numbers in the A and B accumulator and places the unsigned result into the 16-bit D accumulator. This unsigned multiply also allows multiple-precision multiplications.

Long And Short Relative Branches

The MC6809 has the capability of program counter relative branching throughout the entire memory map. In this mode, if the branch is to be taken, the 8 or 16-bit signed offset is added to the value of the program counter to be used as the effective address. This allows the program to branch anywhere in the 64K memory map. Position independent code can be easily generated through the use of relative branching. Both short (8-bit) and long (16-bit) branches are available.

SYNC

After encountering a Sync instruction, the MPU enters a Sync state, stops processing instructions and waits for an interrupt. If the pending interrupt is non-maskable (NMI) or maskable (FIRQ, IRQ) with its mask bit (F or I) clear, the processor will clear the Sync state and perform the normal interrupt stacking and service routine. Since FIRQ and IRQ are not edge-triggered, a low level with a minimum duration of three cycles is required to assure that the interrupt will be taken. If the pending interrupt is maskable (FIRQ, IRQ) with its mask bit (F or I) set, the processor will clear the Sync state and continue processing in sequence. Figure 18 depicts Sync timing.

Software Interrupts

A Software Interrupt is an instruction which will cause an interrupt, and its associated vector fetch. These Software Interrupts are useful in operating system calls, software debugging, trace operations, memory mapping, and software development systems. Three levels of SWI are available on this MC6809, and are prioritized in the following order: SWI, SWI2, SWI3.

16-Bit Operations

The MC6809 has the capability of processing 16-bit data. These instructions include loads, stores, compares, adds, subtracts, transfers, exchanges, pushes and pulls.

CYCLE-BY-CYCLE OPERATION

The address bus cycle-by-cycle performance chart illustrates the memory-access sequence corresponding to each possible instruction and addressing mode in the MC6809. Each instruction begins with an opcode fetch. While that opcode is being internally decoded, the next program byte is always fetched. (Most instructions will use the next byte, so this technique considerably speeds throughput). Next, the operation of each opcode will follow the flow chart. \overline{VMA} is an indication of \overline{FFFF}_{16} on the address bus, $R/\overline{W} = 1$ and $BS = 0$. The following examples illustrate the use of the chart; see Figure 19.

LBSR (Branch taken)

Cycle

1	opcode Fetch
2	opcode +
3	opcode +
4	\overline{VMA}
5	\overline{VMA}
6	ADDR
7	\overline{VMA}
8	STACK (write)
9	STACK (write)

DEC (Extended)

1	opcode Fetch
2	opcode +
3	opcode +
4	\overline{VMA}
5	ADDR (read)
6	\overline{VMA}
7	ADDR (write)

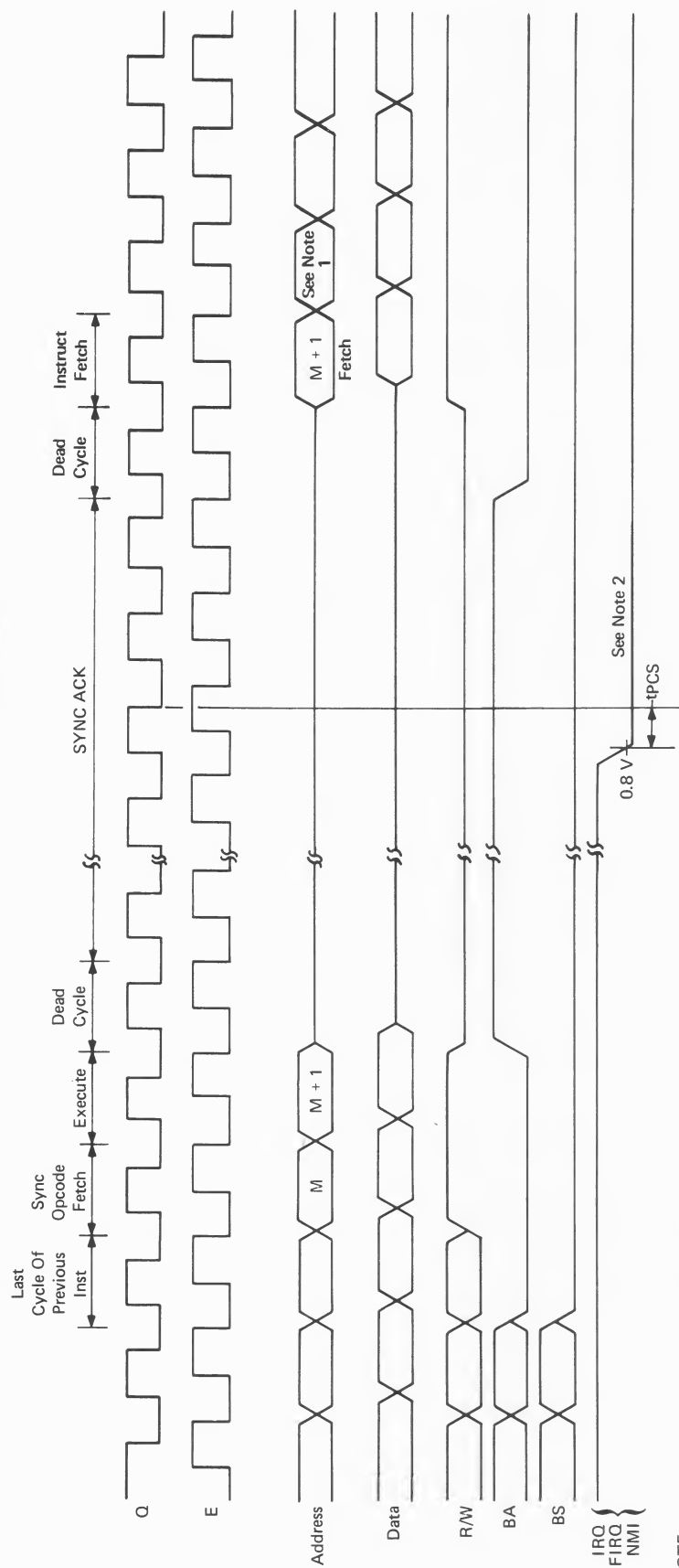
MC6809 INSTRUCTION SET TABLES

The instructions of the MC6809 have been broken down into five different categories. They are as follows:

- 8-Bit operation (Table 4)
- 16-Bit operation (Table 5)
- Index register/stack pointer instructions (Table 6)
- Relative branches (long and short) (Table 7)
- Miscellaneous instructions (Table 8)
- Hexadecimal Value instructions (Table 9)



FIGURE 18 — SYNC TIMING



NOTE:

1. If the mask bit is set when the interrupt is requested processing will continue with instruction execution fetched from previous step. However, if an NMI or an unmasked FIRQ or IRQ caused interrupt, the address placed on bus from previous cycle (M + 1) remains on bus and processing continues with this cycle as (m + 1) or (n + 1) of interrupt timing.
2. If mask bits are clear IRQ & FIRQ must be held low for three cycles to guarantee interrupt to be taken, although only one cycle is necessary to bring the processor out of SYNC.



FIGURE 19 — ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE

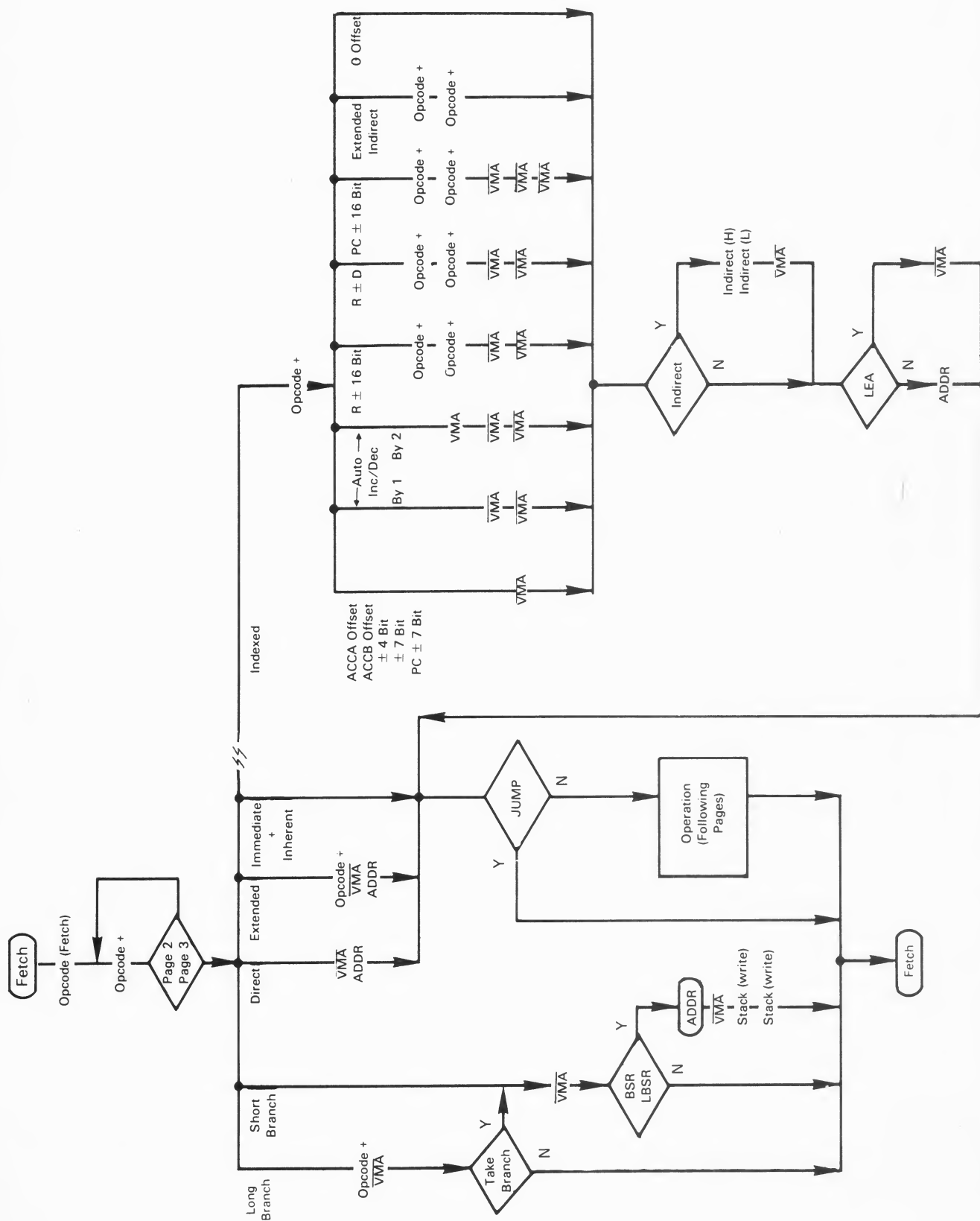


FIGURE 19 — ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE (CONT.)

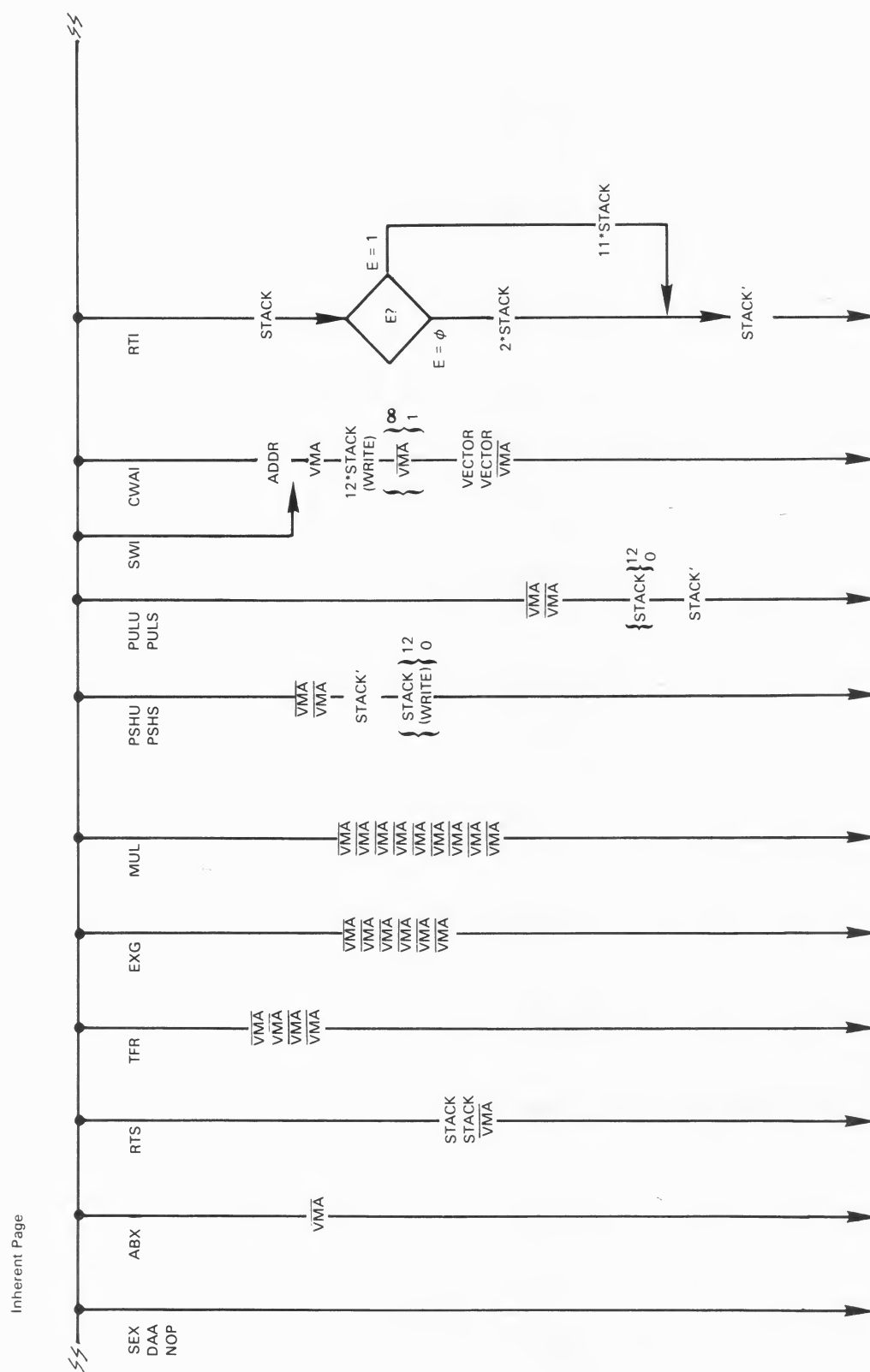


FIGURE 19 — ADDRESS BUS CYCLE-BY-CYCLE PERFORMANCE (CONT.)

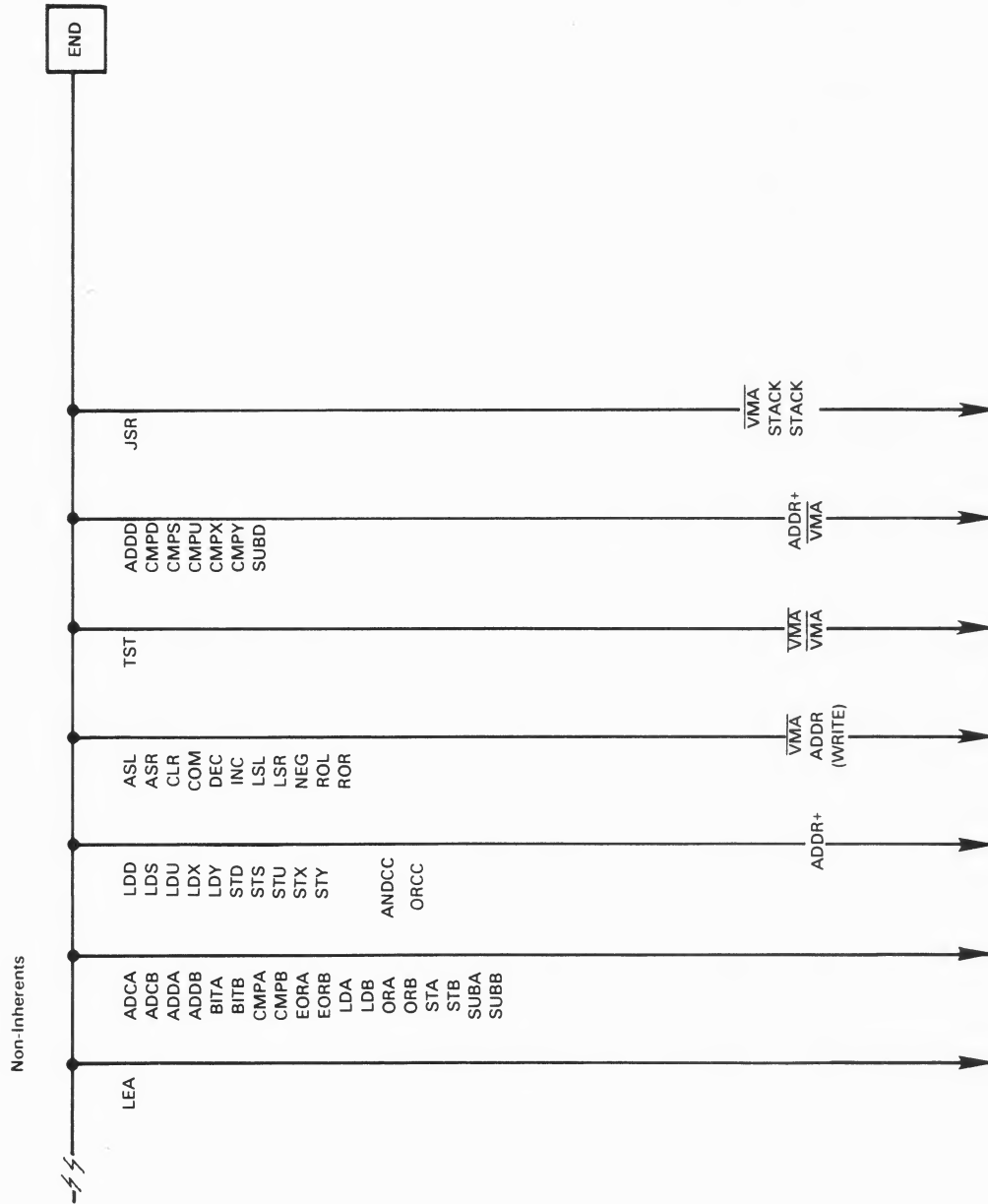


TABLE 4 — 8-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS

Mnemonic(s)	Operation
ADCA, ADCB	Add memory to accumulator with carry
ADDA, ADDB	Add memory to accumulator
ANDA, ANDB	And memory with accumulator
ASL, ASLA, ASLB	Arithmetic shift of accumulator or memory left
ASR, ASRA, ASRB	Arithmetic shift of accumulator or memory right
BITA, BITB	Bit test memory with accumulator
CLR, CLRA, CLRB	Clear accumulator or memory location
CMFA, CMPB	Compare memory from accumulator
COM, COMA, COMB	Complement accumulator or memory location
DAA	Decimal adjust A-accumulator
DEC, DECA, DECB	Decrement accumulator or memory location
EORA, EORB	Exclusive or memory with accumulator
EXG R1, R2	Exchange R1 with R2 (R1, R2 = A, B, CC, DP)
INC, INCA, INCB	Increment accumulator or memory location
LDA, LDB	Load accumulator from memory
LSL, LSLA, LSLB	Logical shift left accumulator or memory location
LSR, LSRA, LSRB	Logical shift right accumulator or memory location
MUL	Unsigned multiply ($A \times B \rightarrow D$)
NEG, NEGA, NEGB	Negate accumulator or memory
ORA, ORB	Or memory with accumulator
ROL, ROLA, ROLB	Rotate accumulator or memory left
ROR, RORA, RORB	Rotate accumulator or memory right
SBCA, SBCB	Subtract memory from accumulator with borrow
STA, STB	Store accumulator to memory
SUBA, SUBB	Subtract memory from accumulator
TST, TSTA, TSTB	Test accumulator or memory location
TFR, R1, R2	Transfer R1 to R2 (R1, R2 = A, B, CC, DP)

NOTE: A, B, CC, or DP may be pushed to (pulled from) either stack with PSHS, PSHU, (PULS, PULU) instructions.

TABLE 5 — 16-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS

Mnemonic(s)	Operation
ADDD	Add memory to D accumulator
CMPD	Compare memory from D accumulator
EXG D, R	Exchange D with X, Y, S, U or PC
LDD	Load D accumulator from memory
SEX	Sign Extend B accumulator into A accumulator
STD	Store D accumulator to memory
SUBD	Subtract memory from D accumulator
TFR D, R	Transfer D to X, Y, S, U or PC
TFR R, D	Transfer X, Y, S, U or PC to D



TABLE 6 — INDEX REGISTER/STACK POINTER INSTRUCTIONS

Mnemonic(s)	Operation
CMPS, CMPU	Compare memory from stack pointer
CMPX, CMPY	Compare memory from index register
EXG R1, R2	Exchange D, X, Y, S, U, or PC with D, X, Y, S, U or PC
LEAS, LEAU	Load effective address into stack pointer
LEAX, LEAY	Load effective address into index register
LDS, LDU	Load stack pointer from memory
LDX, LDY	Load index register from memory
PSHS	Push any register(s) onto hardware stack (except S)
PSHU	Push any register(s) onto user stack (except U)
PULS	Pull any register(s) from hardware stack (except S)
PULU	Pull any register(s) from hardware stack (except U)
STS, STU	Store stack pointer to memory
STX, STY	Store index register to memory
TFR R1, R2	Transfer D, X, Y, S, U or PC to D, X, Y, S, U or PC
ABX	Add B accumulator to X (unsigned)

TABLE 7 — BRANCH INSTRUCTIONS

Mnemonic(s)	Operation
BCC, LBCC	Branch if carry clear
BCS, LBCS	Branch if carry set
BEQ, LBEQ	Branch if equal
BGE, LBGE	Branch if greater than or equal (signed)
BGT, LBGT	Branch if greater (signed)
BHI, LBHI	Branch if higher (unsigned)
BHS, LBHS	Branch if higher or same (unsigned)
BLE, LBLE	Branch if less than or equal (signed)
BLO, LBLO	Branch if lower (unsigned)
BLS, LBLS	Branch if lower or same (unsigned)
BLT, LBLT	Branch if less than (signed)
BMI, LBMI	Branch if minus
BNE, LBNE	Branch if not equal
BPL, LBPL	Branch if plus
BRA, LBRA	Branch always
BRN, LBRN	Branch never
BSR, LBSR	Branch to subroutine
BVC, LBVC	Branch if overflow clear
BVS, LBVS	Branch if overflow set

TABLE 8 — MISCELLANEOUS INSTRUCTIONS

Mnemonic(s)	Operation
ANDCC	AND condition code register
CWAI	AND condition code register, then wait for interrupt
NOP	No operation
ORCC	OR condition code register
JMP	Jump
JSR	Jump to subroutine
RTI	Return from interrupt
RTS	Return from subroutine
SWI, SWI2, SWI3	Software interrupt (absolute indirect)
SYNC	Synchronize with interrupt line



TABLE 9 — HEXADECIMAL VALUES OF MACHINE CODES

OP Mnem	Mode	~	#	OP Mnem	Mode	~	#	OP Mnem	Mode	~	#
00 NEG	Direct	6	2	30 LEAX	Indexed	4+	2+	60 NEG	Indexed	6+	2+
01 *	↑			31 LEAY	↑	4+	2+	61 *	↑		
02 *				32 LEAS		4+	2+	62 *			
03 COM		6	2	33 LEAU	Indexed	4+	2+	63 COM		6+	2+
04 LSR		6	2	34 PSHS	Inherent	5+	2	64 LSR		6+	2+
05 *				35 PULS	↑	5+	2	65 *			
06 ROR		6	2	36 PSHU		5+	2	66 ROR		6+	2+
07 ASR		6	2	37 PULU		5+	2	67 ASR		6+	2+
08 ASL/LSL		6	2	38 *				68 ASL/LSL		6+	2+
09 ROL		6	2	39 RTS		5	1	69 ROL		6+	2+
0A DEC		6	2	3A ABX	↓	3	1	6A DEC		6+	2+
0B *				3B RTI		6/15	1	6B *			
0C INC		6	2	3C CWA		20	2	6C INC		6+	2+
0D TST		6	2	3D MUL		11	1	6D TST		6+	2+
0E JMP	↓	3	2	3E *	↓			6E JMP	↓	3+	2+
0F CLR		6	2	3F SWI		19	1	6F CLR		6+	2+
10 Page 2	—	—	—	40 NEGA	Inherent	2	1	70 NEG	Extended	7	3
11 Page 3	—	—	—	41 *	↑			71 *	↑		
12 NOP	Inherent	2	1	42 *				72 *			
13 SYNC	Inherent	2	1	43 COMA		2	1	73 COM		7	3
14 *	↑			44 LSRA		2	1	74 LSR		7	3
15 *				45 *				75 *			
16 LBRA	Relative	5	3	46 RORA		2	1	76 ROR		7	3
17 LBSR	Relative	9	3	47 ASRA		2	1	77 ASR		7	3
18 *	↑			48 ASLA/LSLA		2	1	78 ASL/LSL		7	3
19 DAA		2	1	49 ROLA		2	1	79 ROL		7	3
1A ORCC	Immed	3	2	4A DECA		2	1	7A DEC		7	3
1B *	↑			4B *				7B *			
1C ANDCC		3	2	4C INCA	↓	2	1	7C INC	↓	7	3
1D SEX	Inherent	2	1	4D TSTA		2	1	7D TST		7	3
1E EXG	↑	8	2	4E *				7E JMP		4	3
1F TFR		6	2	4F CLRA	Inherent	2	1	7F CLR	Extended	7	3
20 BRA	Relative	3	2	50 NEGB	Inherent	2	1	80 SUBA	Immed	2	2
21 BRN	↑	3	2	51 *	↑			81 CMPA	↑	2	2
22 BHI		3	2	52 *				82 SBCA		2	2
23 BLS		3	2	53 COMB		2	1	83 SUBD		4	3
24 BHS/BCC		3	2	54 LSRB		2	1	84 ANDA		2	2
25 BLO/BCS		3	2	55 *				85 BITA		2	2
26 BNE		3	2	56 *				86 LDA		2	2
27 BEQ		3	2	56 RORB		2	1	87 *			
28 BVC		3	2	57 ASRA		2	1	88 EORA		2	2
29 BVS		3	2	58 ASLB/LSLB		2	1	89 ADCA		2	2
2A BPL		3	2	59 ROLB		2	1	8A ORA		2	2
2B BMI	↑	3	2	5A DECB	↓	2	1	8B ADDA	↓	2	2
2C BGE		3	2	5B *				8C CMPX		4	3
2D BLT	↓	3	2	5C INCB		2	1	8D BSR		7	2
2E BGT		3	2	5D TSTB		2	1	8E LDX	Immed	3	3
2F BLE	Relative	3	2	5E *	↓			8F *	↓		
				5F CLR B		2	1				

Legend:

~ Number of MPU cycles (less possible push/pull or indexed-mode cycles)

Number of program bytes

* Denotes unused opcode

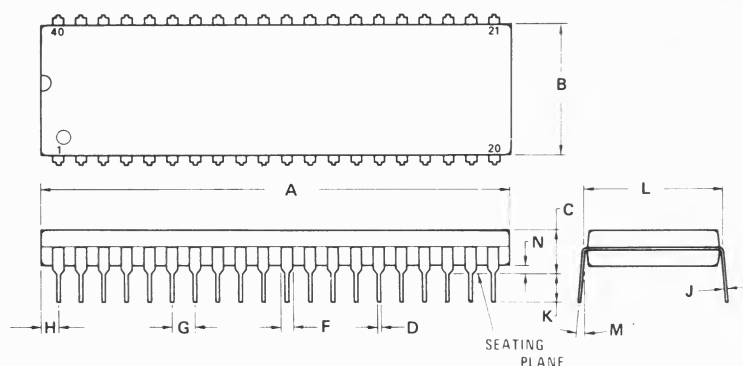


TABLE 9 — HEXADECIMAL VALUES OF MECHANICAL CODES (CONTINUED)

OP Mnem	Mode	~	#	OP Mnem	Mode	~	#	OP Mnem	Mode	~	#
90 SUBA	Direct	4	2	C6 LDB	Immed	2	2	FC LDD	Extended	6	3
91 CMPA	↑	4	2	C7 *	↓			FD STD	↓	6	3
92 SBCA	4	2		C8 EORB	2	2		FE LDU	6	3	
93 SUBD	6	2		C9 ADCB	2	2		FF STU	Extended	6	3
94 ANDA	4	2		CA ORB	2	2					
95 BITA	4	2		CB ADDB	2	2					
96 LDA	4	2		CC LDD	3	3					
97 STA	4	2		CD *	↓						
98 EORA	4	2		CE LDU	Immed	3	3				
99 ADCA	4	2		CF *	↓						
9A ORA	4	2									
9B ADDA	4	2		D0 SUBB	Direct	4	2	1021 LBRN	Relative	5	4
9C CMPX	6	2		D1 CMPB	4	2		1022 LBHI	↑	5(6)	4
9D JSR	7	2		D2 SBCB	4	2		1023 LBLS	5(6)	4	
9E LDX	5	2		D3 ADDD	6	2		1024 LBHS/LBCC	5(6)	4	
9F STX	Direct	5	2	D4 ANDB	4	2		1025 LBBS/LBLO	5(6)	4	
				D5 BITB	4	2		1026 LBNE	5(6)	4	
A0 SUBA	Indexed	4+	2+	D6 LDB	4	2		1027 LBEQ	5(6)	4	
A1 CMPA	↑	4+	2+	D7 STB	4	2		1028 LBVC	5(6)	4	
A2 SBCA	4+	2+		D8 EORB	4	2		1029 LBVS	5(6)	4	
A3 SUBD	6+	2+		D9 ADCB	4	2		102A LBPL	5(6)	4	
A4 ANDA	4+	2+		DA ORB	4	2		102B LBMI	5(6)	4	
A5 BITA	4+	2+		DB ADDB	4	2		102C LBGE	5(6)	4	
A6 LDA	4+	2+		DC LDD	5	2		102D LBLT	Relative	5(6)	4
A7 STA	4+	2+		DD STD	5	2		102E LBGT	Relative	5(6)	4
A8 EORA	4+	2+		DE LDU	5	2		102F LBLE	Relative	5(6)	4
A9 ADCA	4+	2+		DF STU	Direct	5	2	103F SWI/2	Inherent	20	2
AA ORA	4+	2+			↓			1083 CMPD	Immed	5	4
AB ADDA	4+	2+		E0 SUBB	Indexed	4+	2+	108C CMPY	↑	5	4
AC CMPX	6+	2+		E1 CMPB	4+	2+		108E LDY	Immed	4	4
AD JSR	7+	2+		E2 SBCB	4+	2+		1093 CMPD	Direct	7	3
AE LDX	5+	2+		E3 ADDD	6+	2+		109C CMPY	↑	7	3
AF STX	Indexed	5+	2+	E4 ANDB	4+	2+		109E LDY	↑	6	3
				E5 BITB	4+	2+		109F STY	Direct	6	3
B0 SUBA	Extended	5	3	E6 LDB	4+	2+		10A3 CMPD	Indexed	7+	3+
B1 CMPA	↑	5	3	E7 STB	4+	2+		10AC CMPY	↑	7+	3+
B2 SBCA	5	3		E8 EORB	4+	2+		10AE LDY	↑	6+	3+
B3 SUBD	7	3		E9 ADCB	4+	2+		10AF STY	Indexed	6+	3+
B4 ANDA	5	3		EA ORB	4+	2+		10B3 CMPD	Extended	8	4
B5 BITA	5	3		EB ADDB	4+	2+		10BC CMPY	↑	8	4
B6 LDA	5	3		EC LDD	5+	2+		10BE LDY	↑	7	4
B7 STA	5	3		ED STD	5+	2+		10BF STY	Extended	7	4
B8 EORA	5	3		EE LDU	5+	2+		10CE LDS	Immed	4	4
B9 ADCA	5	3		EF STU	Indexed	5+	2+	10DE LDS	Direct	6	3
BA ORA	5	3			↓			10DF STS	Direct	6	3
BB ADDA	5	3		F0 SUBB	Extended	5	3	10EE LDS	Indexed	6+	3+
BC CMPX	7	3		F1 CMPB	↑	5	3	10EF STS	Indexed	6+	3+
BD JSR	8	3		F2 SBCB	5	3		10FE LDS	Extended	7	4
BE LDX	6	3		F3 ADDD	7	3		10FF STS	Extended	7	4
BF STX	Extended	6	3	F4 ANDB	5	3		113F SWI/3	Inherent	20	2
				F5 BITB	5	3		1183 CMPI	Immed	5	4
C0 SUBB	Immed	2	2	F6 LDB	5	3		118C CMPS	Immed	5	4
C1 CMPB	↑	2	2	F7 STB	5	3		1193 CMPI	Direct	7	3
C2 SBCB	2	2		F8 EORB	5	3		119C CMPS	Direct	7	3
C3 ADDD	4	3		F9 ADCB	5	3		11A3 CMPI	Indexed	7+	3+
C4 ANDB	2	2		FA ORB	5	3		11AC CMPS	Indexed	7+	3+
C5 BITB	Immed	2	2	FB ADDB	Extended	5	3	11B3 CMPI	Extended	8	4
					↑			11BC CMPS	Extended	8	4

NOTE: All unused opcodes are both undefined and illegal.





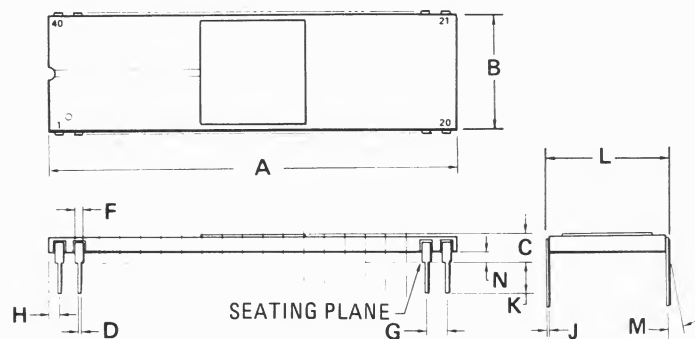
NOTES:

1. POSITIONAL TOLERANCE OF LEADS (D), SHALL BE WITHIN 0.25 mm (0.010) AT MAXIMUM MATERIAL CONDITION, IN RELATION TO SEATING PLANE AND EACH OTHER.
2. DIMENSION L TO CENTER OF LEADS WHEN FORMED PARALLEL.
3. DIMENSION B DOES NOT INCLUDE MOLD FLASH.

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	51.69	52.45	2.035	2.065
B	13.72	14.22	0.540	0.560
C	3.94	5.08	0.155	0.200
D	0.36	0.56	0.014	0.022
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.38	0.008	0.015
K	2.92	3.43	0.115	0.135
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.02	0.020	0.040

P SUFFIX

PLASTIC PACKAGE
CASE 711-03



NOTES:

1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX MAT'L CONDITION.
2. DIMENSION "L" TO CENTER OF LEADS WHEN FORMED PARALLEL.

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.94	15.34	0.588	0.604
C	3.05	4.06	0.120	0.160
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.99	15.49	0.590	0.610
M	—	10°	—	10°
N	1.02	1.52	0.040	0.060

L SUFFIX

CERAMIC PACKAGE
CASE 715-03

Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.



MOTOROLA Semiconductor Products Inc.

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

POSITIVE POWERS OF 2

n	2^n	
0	1	
1	2	
2	4	
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1024	
11	2048	
12	4096	
13	8192	
14	16384	
15	32768	
6	65536	
7	13107	2
18	26214	4
19	52428	8
20	10485	76
21	20971	52
22	41943	04
23	83886	08
24	16777	216
25	33554	432
26	67108	864
27	13421	7728
28	26843	5456
29	53687	0912
30	10737	41824
31	21474	83648
32	42949	67296

NEGATIVE POWERS OF 2

n	2^{-n}					
0	1.0					
1	0.5					
2	0.25					
3	0.125					
4	0.0625					
5	0.03125					
6	0.01562	5				
7	0.00781	25				
8	0.00390	625				
9	0.00195	3125				
10	0.00097	65625				
11	0.00048	82812	5			
12	0.00024	41406	25			
13	0.00012	20703	125			
14	0.00006	10351	5625			
15	0.00003	05175	78125			
16	0.00001	52587	89062	5		
17	0.00000	76293	94531	25		
18	0.00000	38146	97265	625		
19	0.00000	19073	48632	8125		
20	0.00000	09536	74316	40625		
21	0.00000	04768	37158	20312	5	
22	0.00000	02384	18579	10156	25	
23	0.00000	01192	09289	55078	125	
24	0.00000	00596	04644	77539	0625	
25	0.00000	00298	02322	38769	53125	
26	0.00000	00149	01161	19384	76562	5
27	0.00000	00074	50580	59692	38281	25
28	0.00000	00037	25290	29846	19140	625
29	0.00000	00018	62645	14923	09570	3125
30	0.00000	00009	31322	57461	54785	15625
31	0.00000	00004	65661	28730	77392	57812 5
32	0.00000	00002	32830	64365	38696	28906 25

POSITIVE POWERS OF 8

n	8^n
0	1
1	8
2	64
3	512
4	4096
5	32768
6	262144
7	2097152
8	16777216

POSITIVE POWERS OF 16

n	16^n
0	1
1	16
2	256
3	4096
4	65536
5	1048576
6	16777216
7	268435456
8	4294967296

NEGATIVE POWERS OF 16

n	16^{-n}
0	1.0
1	0.0625
2	0.00390625
3	0.000244140625
4	0.0000152587890625

